# On Improving Cybersecurity Through Memory Isolation Using Systems Management Mode

A thesis submitted for the degree of Doctor of Philosophy

James Andrew Sutherland
School of Design and Informatics
University of Abertay Dundee

August 2018

# Declaration

## Candidate's declarations

I, James Andrew Sutherland, hereby certify that this thesis submitted in partial fulfilment of the requirements for the award of Doctor of Philosophy (PhD), Abertay University, is wholly my own work unless otherwise referenced or acknowledged. This work has not been submitted for any other qualification at any other academic institution.

Signed [redacted]
Date Friday, 31st August 2018

## Supervisor's declaration

I, R Ian Ferguson, hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of Doctor of Philosophy (PhD) in Abertay University and that the candidate is qualified to submit this thesis in application for that degree.

Signed .....
Date Friday, 31st August 2018

## Certificate of Approval

I certify that this is a true and accurate version of the thesis approved by the examiners, and [redacted] ordinance regulations have been fulfilled.

Supervisor..[redacted].....
Date.....

# Abstract

This thesis describes research into security mechanisms for protecting sensitive areas of memory from tampering or intrusion using the facilities of Systems Management Mode.

The essence and challenge of modern computer security is to isolate or contain data and applications in a variety of ways, while still allowing sharing where desirable. If Alice and Bob share a computer, Alice should not be able to access Bob's passwords or other data; Alice's web browser should not be able to be tricked into sending email, and viewing a social networking web page in that browser should not allow that page to interact with her online banking service.

The aim of this work is to explore techniques for such isolation and how they can be used usefully on standard PCs.

This work focuses on the creation of a small dedicated area to perform cryptographic operations, isolated from the rest of the system. This is a sufficiently useful facility that many modern devices such as smartphones incorporate dedicated hardware for this purpose, but other approaches have advantages which are discussed.

As a case study, this research included the creation of a secure web server whose encryption key is protected using this approach such that even an intruder with full Administrator level access cannot extract the key. A proof of concept backdoor which captures and exfiltrates encryption keys using a modified processor was also demonstrated.

# Contents

# List of Tables

# Acknowledgements

I would like to express my gratitude to my supervisors, Dr Natalie Coull, Dr R Ian Ferguson and Dr Allan MacLeod for their support and guidance over this process, as well as Dr Adam Sampson, Dr George Weir and Dr John Grigor who performed my *viva voce* examination.

I am also very grateful to my fiancée Emily, my mother and my grandfather for their support in life through my research and beyond.

*—Every contact leaves a trace*

Dr Edmond Locard

# 1 Introduction

It is a founding principle of forensic science that 'Every contact leaves a trace' (Dr Edmond Locard, elaborated by Kirk 1953). More recently, it has become apparent that every point of contact between entities in computer systems leaves analogous traces, often with important security implications.

This makes it important to achieve effective isolation between components, in particular between sensitive ones and those most vulnerable to compromise or subject to external influence; this concept was implemented in OpenSSH by

Provos, Friedl and Honeyman 2003 and is discussed further and a semi-automated mechanism for implementing such isolation introduced in Brumley and Song 2004, including an experiment modifying the `stunnel` SSL/TLS server to handle the keys in a separate process similar to experiment 2b described later in this thesis.

Merely reading a value in memory has long been considered side-effect free for security and developer purposes (Gifford and Lucassen 1986). Processor caching, however, contradicts this: by precisely measuring the time taken to retrieve data from the same location later, other code can determine whether or not that retrieval operation was performed against a given location — leading to the Spectre attacks (Kocher et al. 2018).

To achieve better security, then, we must manage — and limit — the interaction between components much better than we have so far. Covert channels and side channels depend on unexpected interactions; RowHammer (Kim et al. 2014), for example, can be used to achieve privilege escalation (Seaborn and Dullien 2015) via a previously-unexpected interaction between physically proximate memory components. Since there was no correlation between physical and virtual addresses (different processes and the kernel would commingle pages arbitrarily), low-privilege pages could easily be found which happened to be adjacent to highly sensitive system ones, allowing tampering. The same applies between virtual machines and hypervisor control structures. As detailed later, the more coarse-grained the sharing gets, the more limited the avenues of attack become, though any level of shared caching can be an avenue of attack (Liu et al. 2015).

The principle was expanded upon by Paul Kirk (Kirk 1953) in another context decades ago, but is also applicable to computer security in this context: to achieve security, we must carefully limit and control every interaction across security peri-

meters:

> Wherever he steps, whatever he touches, whatever he leaves, even unconsciously, will serve as a silent witness against him. Not only his fingerprints or his footprints, but his hair, the fibers from his clothes, the glass he breaks, the tool mark he leaves, the paint he scratches, the blood or semen he deposits or collects. All of these and more, bear mute witness against him. This is evidence that does not forget. It is not confused by the excitement of the moment. It is not absent because human witnesses are. It is factual evidence. Physical evidence cannot be wrong, it cannot perjure itself, it cannot be wholly absent. Only human failure to find it, study and understand it, can diminish its value. — Paul Kirk (Kirk 1953)

## 1.1 Background

This work began with an investigation of direct memory access attacks, partly inspired by proof of concept attacks over Thunderbolt (Hudson and Rudolph 2015) and FireWire/IEEE1394 (Dornseif 2004). Proposed protection for the system's primary processor against such attacks dates at least as far back as MIT's proposed 2002 memory authentication scheme (Suh et al. 2003). Previous work had focussed on implementing a more limited coprocessor dedicated to security-essential functions (Yee 1994) (Smith and Weingart 1999); other work proposed architectural modifications intended to achieve more secure execution (Lie et al. 2000).

At an early stage, during 2014, some other aspects of hardware isolation and data protection were also investigated, including simulating a processor-level back-

door for capturing and subsequently exfiltrating encryption keys being used in conjunction with Intel's AES acceleration instruction set, AES-NI, as introduced in 2010. Four years after this work was published, it was revealed in *CVE-2018-3665: Intel Core-based Processors 'Lazy FPU Restore' Lets Local Users Obtain Potentially Sensitive FPU State Information on the Target System* 2018 under the name 'LazyFP' that Intel had in fact unintentionally implemented a very similar mechanism in production silicon for years, disclosing encryption keys across process and virtual machine boundaries.

In investigating DMA attacks and protection, it became clear that either DMA would have to be disabled entirely (Microsoft's proposed mitigation at the time, Microsoft Corporation 2017), carefully managed via IOMMU I/O virtualisation (on platforms which support this), or through use of a distinct area of memory which is already excluded from DMA access for other reasons — an enclave. Whilst the former option has significant drawbacks (loss of performance or even functionality on external storage) and the IOMMU was only introduced to x86 as 'VT-d' in Intel's 'Nehalem' microarchitecture, with availability being limited and full usage requiring significant driver modifications. The Qubes operating system and Xen research project on which it built both explored these issues in depth in various ways, creating mechanisms for failure containment on drivers and their associated devices.

Computer security is largely concerned with erecting boundaries between entities: users, privilege levels, processes. Wherever a resource crosses a boundary, it creates the potential for compromise, either through passive information leakage (as in the case of timing attacks, where the exact details of how long an operation takes inadvertently discloses some information) or the potential for active tamper-

ing (as in RowHammer, where writing to one memory location indirectly affects another through non-obvious electrical coupling between parts of a memory chip).

Historically, the term 'enclave' was used to refer to 'a part of a country that is surrounded by another country, or a group of people who are different from the people living in the surrounding area' (Cambridge University Press 2018). In a computer security context, the term is applied to '(a) an isolated process, executed on a platform that provides confidentiality and integrity of code and data as well as (b) sealing and (c) attestation.' (Beekman 2016). In this case, the focus is on (a) and (b) as detailed later under Threat Model, since attestation addresses specific scenarios not applicable in this case.

This work explores various threats and mitigation techniques, then evaluates some possibilities on existing x86 architecture in terms of protecting against various avenues of attack. While the 'enclave' concept is a particular approach to guarding against one threat model with substantial overheads and associated issues, an alternative approach is explored here, aiming to deliver a 'defence in depth' using the existing x86 architecture in novel ways.

Commercial enclave offerings such as Intel's SGX aim to offer a minimal Trusted Computing Base (TCB), in which the security of the system is reliant largely on Intel alone, rather than on the hosting company or the manufacturers of other system components. (For example, any compromise of the RAM could not result in information disclosure, since the contents remain encrypted with a key available only to the CPU or Intel.)

More generally, SGX is one cryptographic technique for reducing the TCB by one component, in this case removing reliance on system RAM and peripherals: code and data are encrypted, guarding against eavesdropping, and checksummed

to guard against tampering. This approach has a long history in computing: network connections are commonly encrypted using TLS for the same reason, Full Disk Encryption and Self Encrypting Drives are increasingly popular for storage.

This protection was defeated in 2018 via side-channel attack (Chen et al. 2018a), forcing Intel to update SGX's deployment mechanism to be able to check the Spectre attacks were properly mitigated on the target hardware.

## 1.2 Context

Within the broader field of computer security, this work combines some existing concepts in a novel way.

### 1.2.1 Privacy and Integrity of Communications

The privacy and integrity of communications has been a high priority in computer science and security research for a long time now, with added financial importance now with the dramatic increase in e-commerce and individual companies in the field now having market capitalisations in the hundreds of billions of pounds.

Security breaches can even have impact beyond the financial, as illustrated by the suicides following the exposure of users of the Ashley Madison adultery facilitation site (Mansfield-Devine 2015).

The scale and importance may be newer, but the underlying need for privacy and protecting information, as well as the use of forms of encryption, dates back at least two thousand years (Churchhouse and Churchhouse 2002).

Within this vast area of study, protection can be divided into three primary areas:

- Avoiding storing information at all — for example, passwords can be kept in a hashed form, enabling correct passwords to be identified without ever storing the password itself.

- Protecting information at rest — Full Disk Encryption and password-protected archive files protect information by requiring a password or other key material before that information can be accessed.

- Protecting information in transit — Protecting messages in transit via cryptography or steganography (both derived from terms for 'hidden writing', using the Greek words κρυπτός and στεγανός respectively).

This work focuses on improving robustness of the last of these areas, through a new approach to the second: protecting encryption keys while held within a computer system for use in securing communications over a network, constructing and analysing a proof of concept web server which handles encryption keys using secure memory protection techniques.

### 1.2.2 Minimum Privilege

Securing a system by denying each principal (a user, hardware or software component) access to anything they do not require access to. These restrictions are an important building block in subsection 1.2.3.

### 1.2.3 Defence in Depth — Fault Containment

In broader engineering terms, it is considered good practice to incorporate fault containment: if and when one component fails for some reason, it should do so in

such a way as to minimise the damage done to the rest of the system. Electrical systems often incorporate fuses and residual current devices, so that when an electrical fault develops it will interrupt the power supply rather than endanger the user or start a fire.

In a computing context, we should consider this for both security and system stability reasons: a malfunctioning word processor should endeavour to shut down without corrupting the files being edited; however badly that word processor might malfunction, the operating system should ensure it cannot damage the file system or interfere with other applications or users on that system. Early implementations were poor at this: a malfunctioning application could hang or crash the whole system, sometimes causing the file system to be corrupted in the process, an issue addressed by process and user isolation and journalled file systems.

### 1.2.4 Minimal Trusted Computing Base

A similar concept to subsection 1.2.2, systems should be designed so that a minimal subset of components have to function properly for security to be maintained. A faulty or malicious mouse, for example, should not be able to leak your password or banking information, even if compromised by a skilled adversary.

## 1.3 Aim & Hypothesis

The aim is to explore the available facilities for data protection within modern processor architectures, including recent special-purpose additions, and more formally:

Demonstrate that secure isolation can be practically implemented us-

ing only the long-established Systems Management Mode mechanisms, giving better security isolation than existing techniques such as process separation.

More specifically, a series of experiments will develop and explore protection of private encryption keys in a server environment, using an HTTPS web server implementation as a test case.

In the chapter 2, the technical background to this work is established, along with the historical context and the details of relevant attack and protection techniques. An analysis of how security can best be achieved in this context is discussed in chapter 3, with the technical details of Systems Management Mode for this purpose in chapter 4.

The experimental technique is discussed in chapter 5. Results of these experiments are then detailed in chapter 6, with an analysis of the implications of these results in chapter 7.

The work is then summarised in chapter 8 followed by a discussion of some possible future work building on this research.

# 2

# Literature Review

## 2.1 Introduction

Memory protection is an essential element of modern operating system and security design. This chapter provides a historical context to the evolution of this concept and modern implementations thereof, as well as highlighting the most important issues discussed later.

From the earliest days of time-sharing systems, research has developed a vari-

ety of schemes for protecting system memory, primarily from internal software threats (partitioning memory to allow multiple independent uses while minimising conflict), as well as a less common goal of guarding against forms of physical external access, be it malfunctioning or misconfigured hardware or malicious interference.

The conventional model of memory learned by programmers and provided by operating system designers is that RAM is simply a homogenous sequence of bytes or words, *uniformly* accessible, changing only when that location is explicitly written to; the more sophisticated model after studying some operating system theory includes the fact each page of memory can be made off-limits, read-only (with or without the ability to execute the contents as code) or writable. Every aspect of this model has been subverted in various ways, leading to significant attack avenues detailed here.

## 2.2 Historical Context

The very earliest computers such as Colossus executed only pre-determined programmes (Copeland 2010), often co-developed with the hardware on which they ran. In the simple implicit trust model of the time, security was not a concern: code and computer were a monolithic entity, with no concerns about conflicting code since there was no other code present, and peripherals had no autonomy which could cause problems.

This had the unintentional advantage of tamper-resistance, an aspect retained for that reason in some security and safety critical designs today in simple embedded systems.

As the world progressed to the Von Neumann architecture, blending code and data, a hierarchy of trust evolved. Time-sharing and multi-user systems created a need for restrictions: one user's code had to be prevented from consuming resources needed by another, or accessing their data without permission, and more intelligent peripherals created the possiblity of conflicting accesses.

The IBM 709 mainframe introduced Channel I/O in 1957, a precursor of Direct Memory Access I/O described later. The preceding 704 model left I/O to the main processor, via simple read and write instructions[1] — on the 709, tape, printer and card devices could be attached and operate independently of the processor. To manage this, an IBM 766 Data Synchronizer gave programmers control of this process, as well as some degree of error handling, although programming was still a monolithic single-tasking system with no concept of privilege levels or access controls: the programme had full control of the data and attached peripherals while executing.

It was on these architectures, however, that John McCarthy and his colleagues first demonstrated time-sharing execution at MIT (Corbató, Merwin-Daggett and Daley 1962). Their eventual 'Compatible Time Sharing System' and the underlying research were prescient in many ways, with consideration of resource starvation, a form of virtual memory in task swapping and even anticipation of the risks of 'thrashing', but security was not yet a major consideration, perhaps because the concurrent users were all physically co-located at the multiple consoles, although password authentication was incorporated.

[1]More specifically, CPY, Copy and Skip, which would read or write from a specified location chosen by a prior RDS – Read Select – or WRS – Write Select – operation; the only error handling consisted of halting execution and lighting the 'read-write check light' on the operator console if the operation timed out. IBM 1955

The two IBM 7094 machines used for this system were modified in one important way to facilitate this service: two separate banks of memory were installed rather than the usual one. The privileged 'A-core' contained and could only be accessed by the 'kernel' code, invoked via interrupts from user code or hardware. (This replaced the previous work using earlier 704 and 709 machines.)

Mainframes made early use of techniques similar to DMA to offload routine data transfer from expensive processors, dating as far back as the vacuum tube-based IBM 709 mainframe of 1958. Unlike the previous model, the 704, this offloaded I/O tasks to the Model 766 Data Synchronizer (Bell and Newell 1971), which could transfer data to or from peripherals while the central processor continued uninterrupted.

Early PCs featured a component called a DMA controller, an Intel 8237, a simple device which could be instructed to read some number of bytes either from memory or from a device, writing each byte to memory or another device. Although devices accessed in this way gain many of the advantages of DMA, they do not have any control over memory access: only the 8237 itself performs actual DMA operations. On a device level, the connection is more akin to the mainframe 'channel' approach, passively receiving read and write commands.

When later facilities such as the PCI bus enabled true DMA, where devices could read or write memory independently, DMA was prefixed with the otherwise-redundant qualifier "bus mastering", or 'BM-DMA'.

## 2.3 Physical RAM

Even before the transistor, computers had Random Access Memory of some form: fundamentally, a set of words which could be retrieved and modified by address. Some of the very earliest computers used mechanical magnetic drum memory for main memory, in a similar system to modern hard or floppy disk drives, or ultrasonic delay lines - in both cases, accessing a particular unit of memory required waiting for that unit to reach the head again.

Magnetic core memory was invented in 1947 and quickly replaced these approaches, itself being replaced by a solid state counterpart in the 70s. The most common since then has been DRAM, Dynamic RAM, in which a single capacitor is either charged (to represent a 1) or discharged (for 0). This is simple, robust and effective, but since a capacitor's charge dissipates over time, a periodic refresh cycle is required, recharging each 1 before it falls to a low enough voltage to be mistaken for a 0.

## 2.4 Hardware Attacks

In the majority of cases, anyone with physical access to the hardware is assumed to be trusted: they have the ability to replace the hardware in its entirety, or components thereof. Exceptions to this threat model, where for various reasons a physical custodian must also be protected against, present an interesting set of situations and solutions.

### 2.4.1 Cold Boot Attack

In a cold boot attack (Halderman et al. 2009), the entire operating system (and hypervisor, if present) is replaced by power-cycling and booting from alternative media provided by the attacker. This technique has long been used to capture forensic images of target systems for 'static analysis' offline, targetting the persistent storage media rather than RAM contents, but the technique also allows capture of RAM contents including encryption keys and other system state, useful to an attacker or forensic analyst.

### 2.4.2 Rowhammer Attacks

This enables the lowest level attack on memory, RowHammer (Kim et al. 2014) — by repeatedly writing to one location in memory to which the attacker does have access, physically adjacent bits can be changed through electrical coupling.

Seaborn and Dullien 2015 demonstrated this from within a web browser sandbox, using the resulting memory corruption to achieve privilege escalation.

*CVE-2018-1038: Microsoft Windows — Local Privilege Escalation* 2018 demonstrated that modifying even a single bit can completely bypass operating system memory protection. In the latter, the user/system bit on the top level page table entry was wrongly set to 'user', enabling unprivileged code to modify all other page table data and so gain full control of all code and memory.

Two versions of mitigation for this attack were incorporated in Intel's Ivy Bridge architecture. Where supported by the memory, the pseudo target row refresh (pTRR) feature causes specific target rows to be refreshed ahead of schedule to prevent excess leakage corrupting values; in the absence of this targeted mech-

anism, Ivy Bridge chips default to refreshing the DRAM at double the normal rate to achieve similar protection at a cost of 2-4% higher memory access latency due to increased frequency of refresh-access conflicts.

### 2.4.3 Bus Level Attacks

Given full physical control of a target system, memory contents can be retrieved and manipulated in various ways which are not otherwise possible. DMA attacks are a variant of this discussed later, exploiting authorised access for unauthorised purposes, and have spawned a whole field of legitimate applications in forensic analysis, including Live Memory Forensics; Windows versions prior to Windows 10 version 1607, for example, could be debugged and probed directly over a 1394 (Firewire) bus cable (Microsoft Corporation 2018), which facilitates DMA. The Intel Direct Connect Interface (DCI) brought this capability to USB 3 ports (Lauterbach GmbH 2018).

Embedded systems such as games consoles are commonly designed to be less amenable to user debugging and reverse engineering, however, and avoid presenting such powerful avenues of attack. George Hotz gained full memory access on the Sony PlayStation 3 (a privilege escalation or hypervisor escape) by corrupting the hypervisor's memory map through a combined hardware and software attack (Lawson 2010): a piece of low-level software to set up appropriate data structures in memory, then a simple fault-injection device implemented on a field programmable gate array (FPGA, programmable hardware) to interfere with the hypervisor's memory access at a crucial point. When executed successfully, this leaves the user code with the ability to read and write otherwise-protected memory

areas and bypass the hypervisor's access controls.

Bus level access was also exploited against Microsoft's XBox by Huang 2002.

This class of attack is more generally referred to as a 'fault attack' or 'glitching attack' (Bar-El et al. 2006), shown to be a powerful and low-cost avenue of attack even for an attacker with limited resources available (Anderson and Kuhn 1997).

### 2.4.4 Test Port Attack

The JTAG (Joint Test Action Group) port is a standardised mechanism for testing and debugging electronic circuits defined by IEEE 1149.1 and IEEE 1149.7. For devices where the manufacturer wishes to impede the owner altering or reverse-engineering the product, this is usually disabled – for example, Microsoft's Xbox 360 games console disables the JTAG ports in software very early in the boot process (DeBusschere and McCambridge 2012).

In a typical desktop/laptop computer scenario, this is not considered a major vulnerability since the JTAG port is not accessible without dissassembly of the computer. Intel, however, exposed comparable system debug functionality via an externally connected USB port (*CVE-2017-5689: Intel Active Management Technology Authentication Flaw Lets Remote and Local Users Gain Elevated Privileges* 2017) as part of their Active Management Technology. (The port was apparently intended to be internal to the system chassis, which would have reduced the security implications in typical scenarios.)

Given the intentionally unconstrained access such ports give, the only effective mechanism to guard against this is to disable the ports in question, as the Xbox does, or physically protect against access, for example by locating a server in a

suitably secure facility.

Taking full control of the memory bus on modern systems is a difficult undertaking: 64 or more data lines carrying signals at speeds of 1 GHz or more, with extremely precise timing requirements. Huang 2002 explored the costs and practicalities, finding the main memory bus to be prohibitive in that context, though access to the narrower and slower HyperTransport peripheral bus was achievable and sufficient. Similar attacks against other parts such as the I2C bus were detailed by Giller 2015.

### 2.4.5 DMA Attack

The development of FireWire and Thunderbolt interfaces exposed DMA facilities outside the relative physical security of the system chassis enabling an early successful attack on Mac OS X systems over FireWire (Dornseif 2004) and later more complex attacks such as ThunderStrike (Hudson and Rudolph 2015). When successful, these attacks give similar functionality to those in subsection 2.4.3 but using existing components rather than physical changes or additions to the target.

## 2.5 Context Switches and KAISER/KPTI

Most modern processor architectures implement some form of virtual memory mapping (Denning 1970): the memory a user process can access at address `0x10000` may be stored in any arbitrary page of physical memory, or indeed be entirely absent and filled in by the operating system when an attempt is next made to access that, known as a 'page fault'.

To reduce the overhead of loading this mapping from memory, processors gen-

erally feature Translation Lookaside Buffers (TLBs), a set of cached address mappings. (Architectures have varying approaches to this; on MIPS, the operating system explicitly populates TLB entries as needed; x86 and more recent ARM variants populate TLB entries directly within the hardware without OS involvement, while the original ARMv2 had 512 explicit memory mappings within the MEMC1 memory controller chip as Content Addressable Memory.)

On x86, paging is controlled by Control Register 3 (CR3, also referred to as the Page Directory Base Register), which holds the physical address of the top level of the page table. Any write operation to CR3 automatically flushes all TLB entries which have not been marked 'global'.

A 'CR3 reload' is therefore an expensive operation in CPU resources, triggering a series of memory accesses, and required on each transition between different processes.

The Meltdown attack (Lipp et al. 2018) exploits small performance differences as a way of accessing otherwise inaccessible memory contents on vulnerable processors, including Intel's Core and Xeon families and some ARM designs. By using a target byte as the offset into an accessible table in main memory, the corresponding row becomes cached. The fetch operation itself is aborted, having accessed a prohibited address, but the next direct access to that row will succeed, and do so faster than access to the other rows since only that one has been preloaded to cache.

Prior to this discovery, it was thought to be safe for the kernel's memory space to be mapped globally at all times, relying on marking those pages as kernel-only. To prevent the Meltdown attack being used, the kernel pages had to be unmapped and remapped when exiting and reentering kernel mode, a mitigation implemented

on the Linux kernel under the names KAISER or KPTI: the protected pages are not merely off-limits (so their address cannot be used directly), but they have no address at all, usable or otherwise.

The x86 architecture's Systems Management Mode, described in detail in chapter 4, already provided a similar level of isolation and consequently already protected against attacks of this type, as well as having protection from DMA attack which KAISER/KPTI does not. The SMM handler code cannot be invoked except by triggering a hardware System Management Interrupt (SMI), which causes the processor to switch memory maps; ARM's TrustZone transfers control to the 'Secure World' environment via an analogous mechanism, their Secure Monitor Call (SMC) exception.

## 2.6 Summary

Comprehensive memory protection addresses a broad spectrum of modern security threats, including pre-emptively guarding against some classes of unknown issues. This thesis proposes an approach to deliver this protection using SMM (Systems Management Mode) in chapter 4, followed by a discussion of previous related work.

Attacks can broadly be divided into physical (those requiring access to the device itself, for example by connecting or disconnecting components) and logical or remote (by reprogramming or tricking existing components in software alone). As detailed later, SMM can be used to guard against both classes: even Administrator or root level access is not sufficient to compromise SMM isolation, and only the processor and memory chips themselves (and the electrical bus connections between them) have the ability to access the protected SMRAM area.

*—The art of war teaches us to rely not on the likelihood of the enemy's not coming, but on our own readiness to receive him; not on the chance of his not attacking, but rather on the fact that we have made our position unassailable.*

Tzu 6th century BC

# 3

# Security Requirements

## 3.1 Introduction

The previous chapter detailed the importance of memory isolation and classes of attacks involved. This chapter develops these themes and analyses their implications in terms of the possible avenues of attack on an SMM-based security implementation in various circumstances, as well as possible mitigations.

The primary hypothesis to be explored here is that hardware memory isol-

ation, as implemented in a case study using Intel's x86 SMM, can be used to deliver similar protection of cryptographic keys to that in the PlayStation 3's Cell architecture or a dedicated hardware security module (HSM), without hardware requirements beyond a standard x86 computer system. Experimentation will aim to demonstrate and evaluate such an implementation, with a particular focus on use to secure a web server's TLS implementation and the public key material on which it depends, through careful consideration and management of the points of interaction between the sensitive components (the cryptographic code and data in use) and the rest of the system.

The security aim is an instance of defence in depth: not intending to prevent any specific attack, but to reduce the impact of successful attacks in general. The later evaluation will consider this in the context of various known attacks and mitigations, particularly those where separation of components either has been or could have been used to prevent or reduce the impact of a compromise.

## 3.2 Scenario

This work aims to secure a network-connected system against remote or transient physical attack, using a simple web server as the model and endeavouring to protect it against unauthorised information disclosure, in particular the cryptographic keys which are used to authenticate the server to clients. (This is clearly generalisable to securing the authentication material on the client end as well: client cryptographic keys, stored passwords, payment mechanisms.)

Other scenarios exist: games consoles attempt to guard against the system owner accessing or modifying data (for example, to reverse engineer the system,

circumvent copy protection or to cheat), which requires a different approach. Attacks such as those mentioned previously in subsection 2.4.3 remain pertinent there however; in particular, the technical and economic aspects discussed in Huang 2002.

To protect the most sensitive data requires the construction of some sort of containment to which access from all other components is restricted or prevented — but with just enough interaction permitted to enable the intended use of the keys (or other material) in question.

For an SSL/TLS web server, the sensitive data is created as a public/private key pair. As the name implies, the public part of the pair may be freely exported and shared — indeed, it is provided to every client connecting, as part of the initial protocol handshake — while the private key is never to be disclosed to anyone else.

To prove the identify of the server, a CSR (Certificate Signing Request) is generated and signed using the private key; after completion of appropriate checks, a Certification Authority (either one trusted by the general public and the software they use, such as LetsEncrypt, or an internal entity such as the US Department of Defense's internal CA) usually signs that CSR to produce a certificate. Any entity can issue certificates, it is merely a matter of policy which issuers are trusted or not for any given situation; for experimental purposes, a self-issued certificate is equally suitable.

LetsEncrypt uses the ACME protocol to sign CSRs without human intervention, using challenge-response proofs of identity as part of a broader effort to make shorter certificate lifetimes practical, while public Certificate Transparency logs provide an additional safeguard against inappropriate certificate issuance.

## 3.3 Threat Model

Two vital considerations in any threat model are the expected levels of technical resources and physical access available to the attacker. A related parameter is the duration of access, from having transient physical custody of a device (for example, a government agency intercepting a server in transit from the manufacturer) to persistent infiltration of a facility (either through personnel, such as a bribed or undercover employee, or technical, as in a compromised device or covert addition to the network).

At one extreme of the former spectrum lie 'nation state' adversaries: government agencies, expected to have substantial budgets and skill sets with few or no legal constraints in place, often with prior knowledge of attacks not yet known to the academic security community and not motivated by direct financial gain — for example Stuxnet (Langner 2011), where the aim was not to obtain information or perform useful work, but merely to damage a specific SCADA system. The opposite end of this spectrum is occupied by a more typical 'hacker' or botnet, using relatively simple attack techniques to harvest banking credentials, send spam or perform volumetric ('brute force') DDoS attacks (Cooke, Jahanian and McPherson 2005).

Physical access runs a spectrum of levels, from no proximity at all, through the ability to attach some external device such as some external storage, to replacing or modifying internal hardware. Orthogonally, this access may be transient or more persistent. The American NSA and their British counterpart GCHQ have long-running programmes to compromise products from the design stage (Ball, Borger and Greenwald 2013), to intercepting equipment in transit from manufacturers to

customers (Kirk 2013) onwards (Kingsbury 2009). Some of the possible attack scenarios are listed below:

- Replace CPU or memory
- Interfere with memory bus, as in the PS3 'glitching' hypervisor escape
- Install keylogger
- Replace or modify BIOS content
- RCE
- Credential Theft
- Social Engineering
- 'Evil Maid' (Tereshkin 2010)

This work aims to guard the encryption keys against all but the upper extreme of attacker resources: remote attacks including root and kernel level compromise, as well as physical attacks short of replacement or modification of the system itself.

A root compromise would allow monitoring of network traffic while the compromise persists, but gaining access to the key would enable impersonation of the server even after the attacker loses access again, which is why the Heartbleed key exposure vulnerability was considered so serious: the additional protection afforded here reduces the damage by such a compromise.

Given an attacker with the resources to replace or modify the processor itself, however (perhaps through a suitably designed microcode patch, rather than physical circuit changes), security is almost impossible to achieve; the first experiment

described in chapter 5 demonstrates the possibilities for an attacker with such capabilities.

## 3.4 RowHammer

The RowHammer attack allows modification of bits in physically adjacent areas of memory, which could theoretically be used to exfiltrate information from the SMM enclave. Integrity checking (a checksum verified on each entry) would provide some protection against this, while ASLR would make such an attack almost impossible — just shifting the code and data by a small random number of bytes each time the system is booted would mean the attacker was operating blindly (able to flip some bits, but without knowledge of which instruction or piece of data is being affected), while the use of 'canary' values around the code and data would make such an attempted attack very unlikely to go undetected.

Moreover, given sufficient knowledge of the memory arrangement in use, simply adding a single disused row between the SMM code and data area and memory used by the system would frustrate any RowHammer attempt: it would corrupt only that buffer space, with no effect on the SMM area.

Also, on the specific test hardware used for the majority of this experimentation, the DDR2 memory installed is much less susceptible to RowHammer attacks anyway: exploiting this generally requires DDR3 or newer, due to the smaller feature size and faster access.

A similar approach would also be effective against most direct hardware attacks, such as address line glitching: without knowing the exact address to target, a successful attack would be very much more difficult than against a system without

this protection.

## 3.5 Spectre/Meltdown

The most recent memory protection attacks against vulnerable Intel and ARM processor architectures pose two potential threats against an SMM protection implementation.

Firstly, the Meltdown techniques can be used directly to extract otherwise protected data, for example from kernel buffers, by using the address of that data indirectly then observing side-effects of that operation. This is not applicable to SMM code or data, since there is no address which refers to that memory in the first place. This was empirically verified by Eclypsium (2018).

Secondly, the Spectre attacks have been used against system firmware executing in SMM to bypass bounds checks (ibid.) — that issue is avoided entirely in this work by using only fixed size parameters, with no bounds checks or boundaries to be violated.

## 3.6 Summary

The set of relevant attacks discussed in chapter 2 was assessed in the context of this threat model, along with some general principles for mitigating or preventing them. In chapter 4 a specific approach to addressing these using SMM will be described, followed by experimentation in chapter 5 for testing the viability of this approach in an example system.

# 4

# Systems Management Mode for Security

## 4.1 Introduction

This chapter presents an overview of the origins of the Systems Management Mode on Intel processors, with information about its attributes and implementation in particular as relevant to a security context, as well as noting related concepts and developments on other platforms and virtualisation approaches.

The chapter concludes with some discussion of other related approaches and

another application of SMM for security purposes.

The experimentation presented in chapter 5 will build on this analysis to explore the main hypothesis, that SMM can be used to deliver improved security isolation.

## 4.2 Origins

The Systems Management Mode was first introduced on Intel's 80386SL processor, the first low-power variant of the 386 architecture intended for portable use. This mode was introduced as a way for the system BIOS to operate certain low-level components (such as the battery, charging subsystem and screen backlight) without conflicting with the main operating system (in those days, typically MS-DOS, which had no inherent support for mobile devices and limited ability to be extended to support them explicitly).

Prior to this, Intel had implemented In-Circuit Emulation — ICE — mode as a debugging facility on the 80286 (Collins 1997b), allowing the full processor state to be saved and later restored (via the undocumented special-purpose `LOADALL` instruction, opcode `0x0f05` or the later 80386 variant, `0x0f07`). On the Pentium, this was then replaced with a more sophisticated Probe Mode (Collins 1997a). Key attributes of SMM and ICE are compared in Table 4.1.

Superficially SMM and the ICE modes have a lot in common: saving the existing processor state to a special-purpose area of memory, switching execution to a dedicated handler routine, then resuming execution later, transparently to the operating system and applications — similar to normal interrupt handling, but with the addition of transitioning to an alternative memory map with areas

| Parameter | SMM | ICE |
|---|---|---|
| Entry trigger pin | `SMI` | `ICEBP` |
| Trigger opcode | N/A[1] | ICEBP (Collins 1996) (`0xf1`) |
| Exit opcode | `RSM` | `LOADALL` |

Table 4.1: Comparison of SMM and ICE attributes

of memory not accessible otherwise.

## 4.3 SMRAM and privilege levels

The defining characteristic of SMM is that while the processor core is executing code in that mode, it asserts the $\overline{\text{SMIACT}}$[2] output line. This signal is interpreted by the Memory Controller Hub (MCH): when asserted, addresses are decoded differently, enabling access to the otherwise-inaccessible SMRAM area. (Physically, this is just part of the main RAM, but gated by the memory controller to prevent non-SMM access.) In early SMM implementations, the address used was `0xA0000`, which is also used by legacy graphics support: any attempt by non-SMM code to read or write this area will access the video memory instead.

The location of SMRAM is defined by the SMBASE register, initially set to `0x30000` (192 kilobytes from the bottom of the memory space); setting the `G_SMRAME` control flag on the processor's `SMRAMC` (SMRAM control) register puts 128 kilobytes of SMRAM at a base address of `0xA0000`, or 640 kilobytes, while setting `T_EN` (TSEG Enable) grants access to a larger area higher up. The address layout is depicted in Table 4.2.

It is important to note that SMM is *not* a privileged mode of execution as such, despite common references to it as 'ring -1' or 'ring -2' as if it were a more privileged

[2]The overscore denotes an electronic signal which is pulled low when active, rather than high

| Address | Size | Content (normal) | Content (SMM) |
|---|---|---|---|
| 0xF0000 | 64k | BIOS ROM | |
| 0xC0000 | 192k | Device ROM/Upper Memory Blocks | |
| 0xA0000 | 128k | Legacy video | SMRAM |
| 0x00000 | 640k | Legacy (DOS) memory | |

Table 4.2: The x86 processor memory map

alternative to ring 0 in which kernel code executes. For example, Wojtczuk and Rutkowska 2009 refers to "escalation from Ring 3 to SMM" — in reality, SMM code is entered in ring 0, and can transition to a reduced privilege level if desired.

In all cases, access to the SMRAM area is permitted only if the access is by the processor core (as opposed to any other peripheral), and then only if either $\overline{\text{SMIACT}}$ is asserted or the `D_OPEN` control bit in the system chipset is set to permit this. As a result, SMRAM has robust protection against any sort of DMA attack: attempted access from the PCI bus or elsewhere is not valid at any time.

## 4.4 Bootstrapping SMM

As noted earlier, access to the dedicated area of memory SMM uses, the SMRAM, is gated by the memory controller. In order to bootstrap the SMI handler, however, it must be possible to load this memory area before the first SMI instance. This is permitted by the `D_OPEN` control bit in the chipset: when set, this bit permits access to SMRAM without being in SMM. After initialisation is complete, this bit should be cleared and the `D_LCK` (Lock) bit set, rendering all the SMM control registers read-only until the processor is reset.

This should be done very early in the system boot process by the system BIOS before activating any peripherals or executing any other code to prevent malicious

code using SMM as a hiding place; older BIOS implementations often failed to secure the state properly during the boot process, leaving the way open for a variety of SMM rootkits at least as far back as 2009 (Embleton, Sparks and Zou 2013).

Attacks on the code executing within SMM are also a possibility, including the recently discovered Spectre technique to bypass bounds checking as discovered by Eclypsium (2018).

## 4.5 Other uses of SMM

### 4.5.1 Security

Soon after malicious use of SMM's isolation property was demonstrated, more benign uses were found, with HyperGuard (Rutkowska and Wojtczuk 2008) in 2008, HyperCheck (Wang, Stavrou and Ghosh 2010) in 2010, HyperVerify (Ding et al. 2013) in 2013 and a US patent on the concept being granted in 2014 (Barde 2014).

The TrustZone-based Real-time Kernel Protection (TZ-RKP) (Azab et al. 2014) applies the same concepts to an ARM system, using ARM's TrustZone mechanism in place of SMM. (TrustZone was created later, with a 'Secure World' entered by invoking a Secure Monitor Call exception.)

The underlying concept in each case is to generate then periodically verify cryptographic hashes of critical structures or code (in HyperGuard's case, by walking the Page Tables to identify all executable pages marked for supervisor access). At the time, this was not wholly sufficient since the processor could still execute

non-supervisor pages with supervisor privilege; the later development of Supervisor Mode Execution Protection (SMEP) by Intel (Ven et al. 2016) closed this loophole.

The level of privilege at which code executes in x86 Protected Mode is determined by the two least significant bits of the CS (Code Selector/Segment) register, so the code at a single address in memory may normally be executed at any privilege level without modification. This has its origins in the 80286's implementation of Protected Mode, prior to the 80386's introduction of paged virtual memory: as the two mechanisms were orthogonal, prior to SMEP a page could be user writable (ring 3) yet run at kernel privilege (ring 0).

### 4.5.2 Forensics/Surveillance

Wang et al presented a set of SMM based tools for firmware assisted memory capture for forensics applications (Wang et al. 2011). Unlike most capture and analysis tools, this is theoretically undetectable to any guest code or intruder except via 'stolen' clock cycles (which may potentially be disguised as normal throttling/power saving) defeating counterforensic measures — a powerful possible alternative to the usual VM sandbox approach for malware analysis.

## 4.6 SGX

Intel Software Guard Extensions aim to deliver similar benefits within the main processor through architectural extensions, with an encrypted area of main memory rather than one isolated by the memory controller hardware. SGX-Shield (Seo et al. 2017) reviews the main limitations of this implementation and proposes

an implementation of ASLR (Address Space Layout Randomisation, varying the location of memory contents to make attacks more difficult) within this enclave for additional protection from outside interference.

This isolation is a mixed blessing, providing a hiding place for less benign code as well (Schwarz et al. 2017), while failing to protect against variants of the Spectre attack (Chen et al. 2018b).

The TaLoS project (Aublin et al. 2017) has significant similarities to the final experiment in chapter 5, in that it seeks to protect the encryption keys and traffic over an SSL/TLS connection but using SGX rather than SMM to isolate the data in question.

## 4.7 Other Platforms

SMM is not a concept confined to the x86 architecture alone. For obvious reasons of ancestry and compatibility it is shared with the x86-64 (*Intel® 64 and IA-32 ArchitecturesSoftware Developer's Manual*) and Itanium (*Intel Platform Innovation Framework for EFI System Management Mode Core Interface Specification (SMM CIS) v0.9*) architectures. As noted previously, ARM's TrustZone can be used as a substitute for SMM in the same protection methodology as HyperGuard.

The Cell Broadband Engine architecture powering the Sony PlayStation 3 (Murase et al. 2009) is a more interesting variant, with multiple PowerPC processor cores in a single physical chip: one or more Power Processing Elements (PPEs) and multiple Synergistic Processing Elements (SPEs). Each SPE has a small local area of memory called the Local Store; when isolation mode is engaged on an SPE, this memory becomes inaccessible from all other components regard-

less of privilege level: nothing outside that processor core can access that memory later, only pass messages to and from that core to request operations and receive the results.

Early in the PlayStation 3 boot process, encryption keys and code are loaded into an SPE which is then placed in isolation mode before the main hypervisor and operating system code is even loaded; as a result, the encryption keys are robustly protected from extraction by would-be reverse engineers (*PlayStation 3 Secrets*). Consequently, attacks have focussed on compromising the earliest stages of the boot process software, ultimately obtaining the key to sign their own replacement 'level 0' bootloader (*Sony PS3 hacked "for good" —- master keys revealed*) rather than attempting to attack the architecture itself.

## 4.8 Software Alternatives

Various attempts have been made to deliver isolation between memory areas in other ways in recent years:

### 4.8.1 Credential Guard — VM isolation

Microsoft recently released a software-only implementation of a similar approach, Credential Guard, in which authentication keys are held in a dedicated virtual machine running on top of the Hyper-V hypervisor platform. This way, even a kernel compromise of the main operating system is not sufficient to extract credentials for reuse: no more 'Pass The Hash' privilege escalation once a system is compromised. Only a compromise of the underlying hypervisor itself, or the hardware isolation mechanisms, would suffice: a much smaller attack surface compared to the full OS

kernel.

### 4.8.2 Process Isolation

The commercial content delivery network (CDN) Cloudflare has an interesting implementation of TLS/SSL in two respects. First, they offer 'Keyless SSL' (*Keyless SSL: The Nitty Gritty Technical Details*) in which the site's private key is handled remotely. Secondly, the SSL/TLS handling is performed in a separate isolated instance of the Nginx web server (*Incident report on memory leak caused by Cloudflare parser bug*) — an example of defence in depth which ensured that when a bug was found in their HTML parsing implementation, the information disclosed could not include site private keys, unlike with the widespread Heartbleed bug in OpenSSL (*The Results of the CloudFlare Challenge*) — only a kernel or hardware level exploit could have exposed the key, not an application level one.

A version of this technique is also covered in the experimentation detailed in chapter 5.

### 4.8.3 Encrypted Memory

TRESOR (Müller, Freiling and Dewald 2011) demonstrated that a general-purpose computer system can be operated with almost all of main memory encrypted while at rest, albeit with a significant performance penalty, using a modified Linux kernel. There is some overlap with the research this thesis describes: TRESOR uses the processor debug registers as an area of storage which cannot be accessed via DMA. This was intended to protect against DMA attacks, among others, but was not successful in that respect since this cannot protect the associated code:

TRESOR-Hunt (Blass and Robertson 2012) demonstrated a successful attack on this protection, using code injection via DMA - an attack which could not be prevented through software mechanisms alone.

TreVisor (Müller, Taubmann and Freiling 2012) extended the techniques of TRESOR to a hypervisor level in combination with techniques from BitVisor (Shinagawa et al. 2009) to incorporate Intel VT-d (IOMMU) protection from DMA attack.

On other platforms, the ARMORED (Gotzfried and Muller 2013) project applied TRESOR techniques to the Android operating system on ARM architecture processors as a countermeasure to their own FROST (Müller and Spreitzenbarth 2013) attack, which used a cold boot attack to retrieve information from mobile handsets running Android 4.0 despite the disk encryption employed.

### 4.8.4 Encrypted Swap

A cold boot attack can retrieve RAM contents for a brief period after a system is shut down, but the system's virtual memory persists indefinitely after shutdown unless explicitly wiped. To avoid this, keeping that data encrypted is an idea which long predates efforts to encrypt or otherwise protect the RAM, including the encrypted swap space (Provos 2000) extensions to the UVM system (Cranor and Parulkar 1999) originally proposed as an enhancement of the original 4.4BSD approach. The much slower nature of disk storage meant the extra overhead of this encryption was more widely accepted early on.

## 4.9 Summary

SMM, although originally intended to deliver other functionality, is used in a variety of ways relevant to security, both positive (to protect systems and data from intruders or corruption) and negative (to escape detection, conceal compromises and covertly capture information from a target). While SMM itself is a specific Intel or x86 instance of this isolation mechanism, other platforms have analogous mechanisms such as ARM TrustZone and virtualised isolation mechanisms, making the underlying principle more broadly applicable. SMM and TrustZone may be regarded as one extreme on a spectrum of context isolation mechanisms, with direct inline handling of key material as the other extreme (no isolation at all) and other options in between, such as separating key handling in either another virtual machine (as Microsoft does for Credential Guard) or another Unix process (as CloudFlare does) — the experiments described later study the performance aspects of three points on this spectrum.

# 5

# Methodology

## 5.1 Introduction

This chapter details a set of experiments which explore the initial hypothesis, that SMM can deliver improved security isolation compared to other software-only approaches such as process or virtual machine isolation.

The initial work consisted of a proof of concept exploit, section 5.2: a simulated hardware backdoor in the x86 architecture whereby cryptographic keys would be

captured then exfiltrated. This did not involve *memory* protection, but served to explore and validate aspects of the later work as well as delivering a result in itself, demonstrating a mechanism for bypassing security by circumventing the isolation between processes.

For the performance assessment, two approaches are used: first, micro-benchmarks, measuring the individual components involved in transitions to and from SMM and kernel mode in isolation; secondly, to measure the overall performance of a web server using different isolation mechanisms, to be able to compare SMM isolation's performance overhead against versions with no isolation of key handling and one which uses process-level isolation (which would protect against process level compromise, but not a root or kernel level one as SMM isolation does).

The experiments in section 5.3 investigate the performance aspects of using SMM, in particular detailing the performance impact of each transition to and from SMM compared to transitions to kernel space and back, which will be the dominant factor in the overall performance of the SMM-isolated server developed for section 5.7 later.

Finally, a set of three experiments described in section 5.5, section 5.6 and section 5.7 test the performance and functionality of a simple HTTPS server application with three different levels of key isolation: none (a control), process separation, and fully SMM isolated key handling.

## 5.2 Experiment 1: CPU Backdoor

This early stage work simulated a 'backdoored' version of an x86 processor. Intel had recently released the Westmere microarchitecture (Gueron 2010), which added hardware acceleration of the AES cryptographic functions to the preceding Nehalem implementation. At this time, Edward Snowden had just exposed some of the National Security Agency's activities, raising new questions about the integrity of many products, the hypothetical question was posed (Bernstein 2013): 'Thought experiment in malicious chip design: What would be the easiest way for an Intel CPU to leak AESKEYGENASSIST inputs to an attacker?'

This work entailed multiple facets: the details of Intel's AES implementation (particularly, the AESKEYGENASSIST instruction mentioned, which expands the raw AES key to the form used for encryption and decryption), some way of capturing and storing this key even across context switches within the operating system or hypervisor, then constructing a covert channel through which the attacker can retrieve the key later.

The simulation itself was constructed by modifying the Bochs open-source system simulator (Lawton 1996), an instruction by instruction simulation of the x86 and x86-64 instruction set.

Initial key capture and retention was simple: a global state variable, loaded with the key operand whenever the AESKEYGENASSIST instruction was called with a Round Constant of 0. (AES has key sizes of 128, 192 or 256 bits; the first 128 bits are always used for round 0, with the remaining bits if any loaded in round 1; for this proof of concept, 128 bit AES was sufficient, demonstrating an approach which could easily extend to the 192 and 256 bit variants if desired.) In

a physical implementation of this design, the variable would be implemented as a 128 bit register. Since this register is a concealed addition of which the operating system and hypervisor have no knowledge, this state will persist across process and virtual machine boundaries: only a full processor reset would clear it.

Exfiltration was via the floating point instruction set, a design partly inspired by the Pentium FDIV bug (Pratt 1995): Intel's flaw was included on all the original Pentium chips, yielding an incorrect result once per $2^{33}$ divisions, and was not discovered for a significant time. An intentional version triggering much less frequently would be effectively undetectable without specific prior knowledge of the numerical parameters chosen. By triggering only for a specific pair of operands (specifically, $1.91207592522993 \times 2^{306}$ being divided by itself), only one in every $2^{128}$ possible divisions yields an unexpected result — in this case, instead of the correct answer of 1.0, the next 32 bits of the last-used AES encryption key are returned.

FDIV — floating point division — was chosen because it is not only an unprivileged operation any code can execute, but also a basic primitive with a direct counterpart in high level languages, including Javascript: performing this particular division calculation will lead to execution of that precise instruction, on any Javascript implementation from the simplest interpreter to the most sophisticated JIT.

With the target key captured in a non-privileged piece of Javascript, relaying that over the Internet within a URL was trivial, appending it to an image URL (Sutherland, Coull and MacLeod 2014).

Late in the course of this research, it emerged that Intel had in fact incorporated a version of this unintentionally (*CVE-2018-3665: Intel Core-based Pro-*

*cessors 'Lazy FPU Restore' Lets Local Users Obtain Potentially Sensitive FPU State Information on the Target System* 2018): the key was stored in a standard processor register in a way which could later be retrieved by another process or virtual machine despite intervening context switches.

This was an early exploration of the security aspects of the processor hardware, serving to test and refine processor simulation techniques and implementation details.

## 5.3 Experiment 2: Micro Benchmarking

Having experimented with low-level simulations, processor architecture and data transfer between privilege levels and across isolation contexts, the next stage was to investigate the performance on real-world hardware: SMM was documented to provide robust isolation, but could it easily provide sufficient performance for a workable web server implementation?

After prototyping work on the Bochs hardware simulation, a physical target system was required for performance tests. A Lenovo ThinkPad X200 was obtained and loaded with the Libreboot free software project's variant of the open-source Coreboot firmware (*Libreboot*), including its SMI handler code which could then be freely modified in theory. An unmodified ThinkPad T60, with similar hardware but retaining the original manufacturer's BIOS, served as control, backup and development system, allowing testing of SMM code under the Qemu-KVM virtualisation system in conjunction with the related SeaBIOS project (*SeaBIOS*).

The first performance tests focus on comparing the raw latency penalty imposed by the architecture on transitions between userspace and either kernel mode

or SMM as appropriate. This would give an early indication of the viability of the overall approach to explore later, as well as determining how much effort might be required to optimise the design for performance to be viable.

Each test consists of executing the function under test multiple times, recording the elapsed time and calculating the time per iteration from that. To ensure consistency, each test was repeated multiple times and checked for outliers.

Timing is measured in two ways: the system 'time of day' clock (which records times in microseconds) and, for the T60 and virtualised system, the processor Time Stamp Counter (read via the `RDTSC` instruction). On recent Intel processors, including those in use here, the time stamp counter advances at a constant rate regardless of power saving modes or clock speed, making this a useful timing measurement. (On earlier implementations, the TSC rate varied with processor speed, making this usage more problematic.)

The operations tested are listed in Table 5.1. Each set of measurements was performed on each test system, to provide a baseline for interpreting performance figures later.

The source code for the tests is included in B, with details of the processor (E), Linux kernel (D) and compiler version (C) used in the tests. The test code was compiled with level 2 optimisation ('-O2'), for x86-64, in each case. To gather statistical details about the distribution of each individual operation, the test code optionally records the TSC value after each; for the overall operations, to avoid the extra overhead, a consecutive sequence of runs is timed without recording timestamps in between, by compiling with the `BATCHONLY` flag. (For the 1,000,000 iterations of `getpid`, 8,000,000 bytes of values are written out to memory, almost four times the size of the L2 cache, although writing the values to disk is deferred

| Operation | Purpose |
|---|---|
| NOP SMI | Round trip to/from SMM |
| `open-close` | System call requiring access to kernel memory |
| `getpid` | Trivial system call to reflect minimal kernel transition cost |

Table 5.1: Operations tested in micro-benchmarking

| Model | X200 | T60 | Qemu-VM |
|---|---|---|---|
| CPU | Core 2 Duo P8400 | Core 2 Duo T5600 | Core 2 Duo T5600 |
| Clockspeed | 2.26 GHz | 1.83GHz | 1.83GHz |
| RAM | 4 GiB | 3 GiB | 1 GiB |
| BIOS | Libreboot | Lenovo original | SeaBIOS |

Table 5.2: Test platforms for benchmarking

until after the timed portion.)

(Ordinarily the `getpid` function is accessed via vDSO for performance reasons — the kernel puts a copy of the PID in the process's own memory space and provides a function to retrieve that directly, avoiding the userspace-kernel round trip, but in order to measure that round trip the legacy system call is used here.)

The resulting timing figures are shown in section 6.2.

## 5.4 SMRAM and SMM

The overall goal is to implement a proof of concept server, in which cryptographic secrets are protected within an SMM enclave, bearing in mind the needs for efficiency and security: in particular, minimal overhead in each transition to/from SMM, and presenting a minimal attack surface on the SMM component while enabling the application counterpart to run with minimal privileges.

As noted earlier, the Sony PlayStation 3 uses a similar approach to isolating its cryptographic keys: one of the Cell's Synergistic Processing Elements is isolated

from the rest of system early in the boot process, functioning as a cryptographic oracle exchanging messages with the rest of the system.

From the programmer's perspective, this enclave will function in many ways akin to a physical hardware device, passing messages in both directions via a page of physical memory.

The starting point is a conventional web server, running as a normal unprivileged application ('ring 3') under Linux.

Entry to SMM requires triggering an SMI (System Management Interrupt). Ordinarily, hardware interrupts cannot be triggered directly from user mode applications; first a system call would be required, to effect a transition to kernel mode ('ring 0' on x86), then the corresponding kernel code would trigger the interrupt on the application's behalf. This, however, incurs additional overhead, two mode transitions rather than one.

A more efficient approach is for the application to write to the I/O address `0xb2` as explained below.

Most modern processors implement a unified hardware memory map, in which RAM and devices occupy the same address space; x86 has two distinct memory spaces, a 64 kilobyte legacy space accessed via the `IN/OUT` set of instructions, and a much larger space accessed via standard memory operations.

For devices mapped into the main memory space, the usual memory permissions apply: the appropriate 4 kilobyte (or larger) page could be mapped with appropriate permission bits set. The I/O space has different, fine-grained permissions: the IOPB (I/O Permissions Bitmap) within the TSS (Task State Segment) controls whether access is granted or not to any given byte within the I/O address space. On Linux, the `ioperm` system call may be used to enable access to any

specified I/O address.

To make use of the cryptographic enclave services, the userspace code must first allocate and lock a page of physical memory, determining the underlying physical address via the Linux `/proc/self/pagemap` virtual file and communicating this to the SMM enclave at initialisation time. This shared page can then be used as a mailslot for exchanging data: the userspace (ring 3) code interacts directly with the SMM cryptographic code, without transitions to/from the kernel in between.

## 5.5 Experiment 3a: Protocol Verification

Through the use of Google's established BoringSSL variant of the open-source OpenSSL project codebase, much of the protocol-handling work required to deliver a functional implementation of HTTPS was simplified. Other implementations including the original OpenSSL were experimented with early on, but BoringSSL is simplified and well-maintained making it better suited here, as CloudFlare concluded for a large-scale use case with some similar elements to this work, notable the private key handling callback mechanism as opposed to OpenSSL's 'engine' approach: *Make SSL boring again.*

As a minimum, any server must have a signed 'certificate' which identifies the server name and the cryptographic public key to be used for communication. In a typical public deployment, the certificate will be signed by a CA such as Comodo or GlobalSign which is already trusted by the common web browsers: the HTTPS server at `2a00:1450:4009:80f::2004` presents a certificate asserting that it is 'www.google.com', GlobalSign sign the certificate asserting that this is verified[1],

[1]In fact, GlobalSign's certificate 'GlobalSign Root CA R2' merely asserts that 'Google In-

and web browsers then trust this assertion as a result.

For experimental purposes, a 'self-signed' certificate is sufficient: the cryptographic mechanisms are identical, but most client applications will alert the user to the unknown identity of the server before proceeding.

For public demonstration and early testing purposes, two of the three most popular web browsers provide good examples (Google Chrome has held a substantial lead in popularity throughout the duration of this project, while Microsoft's Internet Explorer and Mozilla Firefox have alternated between second and third place).

For automated protocol testing, the standard Unix utilities `curl` and `wget` are both well-established, with the former in particular being noted for protocol support: early support for the new HTTP2 protocol and Brotli compression algorithm, as well as easy integration into various other platforms via the `libcurl` library and associated language bindings.

Once basic functionality was established, more extensive protocol functionality and compatibility testing is available via a variety of published testing tools. A combination of two established test suites, the Qualys SSL Labs tool (Qualys 2014) and `textssl.sh` (Wetter 2016) provide a substantial and well-regarded array of tests for both compatibility with a variety of simulated client applications and platforms and testing for various common implementation errors, such as accepting weak or invalid cryptographic keys, obsolete algorithms or allowing inappropriate key reuse. During development, the Wireshark packet capture/analysis tool was also invaluable for debugging interoperability issues between the test server and

ternet Authority G3' can be trusted, and that certificate in turn validates that in use on www.google.com.

client applications.

The goal of this experiment was to achieve a working server with demonstrated broad client compatibility, through a combination of manual client application testing and the automated test suites.

### 5.5.1 Tool: Wireshark

Wireshark is an established open-source network traffic capture and analysis tool, originally released under the name Ethereal around 1998. The graphical interface now features advanced packet dissection tools, advocated for TCP/IP teaching in lab sessions (Wang, Xu and Yan 2010) — as it now includes the ability to interpret SSL/TLS message structures in the same way, this makes it an excellent diagnostic and testing tool for this experiment as well.

### 5.5.2 Tool: Qualys

The protocol parameters and configuration were largely chosen in accordance with Qualys 2014 for best practice (with certain justified deviation: for example, OCSP stapling and HTTP Strict Transport Security are not applicable when not using publicly trusted certificates, and both TLS session reuse and HTTP keepalives are performance enhancements which would just impede the performance testing later). These configuration details and a set of known common issues are embodied in the Qualys SSL Labs test suite, along with a system for simulating the cryptographic handshake processes used by a variety of HTTP client applications and host platform versions (for example, different versions of Google Chrome on Windows XP, Windows Vista and others).

## 5.6 Experiment 3b: Process Isolated Key Handling

This experiment modified the initial test bed HTTPS server to segregate the cryptographic private key used outside the main server process. As noted earlier, this separation provides some additional security protection (a compromise of the main server application, as in Heartbleed, would no longer be sufficient to expose the server key) akin to that presently deployed commercially by CloudFlare, reflecting an intermediate position on the spectrum between the full isolation of SMM or other mechanisms and unprotected key handling.

No new tooling was involved in this experiment, just additional coding (using a page of shared memory and two file descriptor pairs from the `pipe` system call to synchronise the interaction between the two processes) and re-running the previous experiment's interoperability tests to guard against regressions. (Any change in code potentially introduces new flaws; ensuring at each stage that the code under test is still a fully functioning SSL/TLS implementation, rather than 'completing' the benchmark without correctly performing all the steps involved.)

## 5.7 Experiment 3c: HTTPS Performance Testing

This experiment tests the full spectrum of key handling options, including SMM hardware-assisted key isolation, and studies the performance impact of using each. In particular, measuring the number of requests per second handled in each con-

figuration to identify the additional overhead contributed by the use of SMM to isolate the cryptographic private key and associated code compared to the control option (no isolation at all) and the simple option (using a separate user-space process for isolation).

To isolate the cryptographic aspects, the simplest possible HTTP situation is ideal: a single very small piece of static content, being requested as many times as possible per second with each request over a fresh encrypted connection using a fresh session key.

In a 'real' web server situation, multiple requests will normally be sent over a single connection, and the client and server will agree to re-use the existing keys from a previous connection rather than perform a fresh handshake each time. This will amortise the overhead of the initial cryptographic handshake across multiple connections. (The server may also be hosting multiple web sites with distinct keys, although major installations such as Google and CloudFlare share keys and certificates across hostnames.)

Both of these factors make the measurements from this experiment a 'worst case' scenario for the performance impact of SMM key isolation, leaving room for improvement as discussed later.

The technical details of the overall design and operation of the server code are described later in this chapter.

To quantify the performance impact of the Qemu-KVM virtualisation system used to test the SMM version, the other two tests take place on both the bare metal test system and under Qemu-KVM. Each performance test is run at four different response sizes: 1k, 10k, 100k and 1MiB, to determine the effect this has. The four sizes were chosen to cover a broad range of possible scenarios, including

| ID | Description |
|---|---|
| 0 | Control - keys handled directly in-process, no isolation |
| 1 | Process separation - keys handled in separate worker process |
| 2 | SMM separation - keys handled in SMM |

Table 5.3: Configurations tested in experiment 3

an extreme case of very small payloads in which the initial connection setup and handshake will dominate, up to a size more likely to be representative of real web content (MachMetrics 2018): a total page size of 2 MiB will be comprised of multiple objects.

### 5.7.1 Tool: http_load

The free http_load tool from ACME (*http_load*) provides a simple HTTP load generator with optional support for HTTPS.

## 5.8 Server Implementation

The long-established OpenSSL (originally SSLeay, by Eric Young) library provides all the code necessary for both client and server SSL/TLS implementations, and had been extended with 'engine' functionality for applications to make use of hardware acceleration devices as well.

After some high-profile security issues highlighted long-standing maintenance and code hygeine issues with this codebase, two forks were created to provide a better service, LibreSSL (under the auspices of the OpenBSD project) and BoringSSL (under Google). Of particular relevance to this project is the addition of `SSL_PRIVATE_KEY_METHOD`, allowing application code to make indirect use of

a private key to which it does not have access – precisely the facility the SMM enclave is intended to deliver.

Combining the BoringSSL library and the SMM cryptographic enclave creates a single simple HTTPS-capable web server which can operate in three modes: handling the public key cryptography and keys in-process (the 'traditional' approach used by most web servers today, where any server compromise exposes the private key as in the Heartbleed bug), across a conventional Linux process boundary (akin to the privilege separation added to OpenSSH in 2002 (Provos, Friedl and Honeyman 2003), giving some protection) or using the SMM enclave (in which even a root or kernel level compromise does not expose the private keys: the keys are generated within SMM and never exposed outside this enclave). The relative performance of all three modes can then be tested and compared, with minimal artefacts from the difference between implementations.

## 5.9 Enclave Implementation

To create the SMM enclave itself, the code must be loaded early in the system boot process. (Specifically, at power-up, the protected SMRAM area can still be configured and accessed by enabling the `D_OPEN` bit, until the `D_LOCK` bit is set to prohibit further access from outside SMM.)

Once loaded, this code is entered whenever the SMI (Systems Management Interrupt) is triggered. On entry, the existing register contents are all saved within the SMRAM State Save Map, so invoking SMI after loading the physical address of the mailslot page into a register is sufficient to establish communication. (A single 4k page is ample for passing signatures and public keys.)

## 5.10 Certificate Signing

For a web server to be accepted as 'valid' for a given name, it must present a signed certificate asserting ownership of that name, signed by either a trusted root Certificate Authority (CA) directly, or an intermediate certificate which is itself trusted.

This is a two stage process. First, a Certificate Signing Request must be generated, containing a copy of the server's public key and a signature using the private key (the private key itself is never exposed).

Secondly, this CSR must be submitted to and accepted by the CA. Originally, this was done manually using human verification of documents and credentials; this still applies for 'Extended Validation' certificates, but for standard 'Domain Validation' certificates this process can now be entirely automatic.

Specifically, the free LetsEncrypt CA allows ownership of a name to be verified by publishing specific challenge response values in the DNS entries of the name in question, without the server ever having to be publicly accessible. (This is one variant of the ACME — Automated Certificate Management Environment — protocol; other variants use the TLS SNI handshake process and HTTP messages respectively to accomplish similar results via other protocols.)

This allows a public-private keypair to be generated within the SMM enclave, issued with a valid certificate, then used to host a secured website for testing and demonstration purposes, without ever exposing the key material externally.

For testing purposes, however, this external signing step is not necessary: a 'self-signed' certificate is sufficient.

## 5.11 Performance Testing

As the cryptographic code is unmodified – a standard x86/x86-64 implementation of the elliptic curve algorithms – the key performance metric is the additional overhead introduced by transitions to and from SMM.

For context, this should be compared with the overhead entailed in a context switch between usermode processes (as applies where the cryptographic code is run in a separate process, as CloudFlare does in their content delivery network's edge devices) and user-kernel mode transitions (particularly after implementation of the KPTI changes to mitigate the Spectre/Meltdown security issues). Experiments 2 and 3c quantify these.

For a better indication of the real-world performance impact, standard HTTPS benchmarking — downloading static content over encrypted connections in each configuration tested — gives indicative throughput speeds.

## 5.12 Security Testing

While web server performance testing is a well studied and long-established field (Trent and Sake 1995, Banga and Druschel 1999), security is more nebulous. In this context, the architecture is intended to provide isolation, and substantial literature has already studied the various possible routes to accessing SMRAM (Wojtczuk and Rutkowska 2009 — cache aliasing, MTRR manipulation; Furtak et al. 2014 — early BIOS implementations which neglected to enable `D_LOCK` timeously). It can also be verified empirically that the SMRAM-protected data/code is not exposed, even to the kernel (via a scan of the Linux `/dev/mem` device, which can

be configured to expose the kernel's view of the entire memory space). Since the SMM protected data has no functioning address except while the processor is executing in SMM, exploits such as Spectre cannot access this data. (Physical level attacks such as RowHammer or address line fault injection could still be effective; potential countermeasures to such attacks are discussed in section 3.4.)

## 5.13 Summary

A set of experiments is described which will serve to explore the functionality of SMM in this context and test the hypothesis presented earlier, while also quantifying the performance impact of each approach.

# 6

# Results

## 6.1 Introduction

This chapter describes the experimental results obtained, with an explanation of the testing methodology used to obtain them.

### 6.1.1 Processor Level Backdoor

This experiment successfully demonstrated a proof of concept simulation of a processor-level backdoor, exploitable from unprivileged code (including Javascript executing within an unprivileged sandbox within a web browser) yet undetectable without specific prior knowledge of the backdoor details.

For demonstration purposes, a file was encrypted using the command line OpenSSL tools on an unmodified Windows XP virtual machine (as provided by Microsoft). Cryptographic verification of the system confirmed it was unmodified, and malware scans were negative, as the only modification was to the simulated processor core itself.

After the encryption operation had been performed, the demonstration web page was loaded in the Internet Explorer web browser. The page loaded normally, with no abnormalities visible to the user — but the encryption key used in the previous stage was now shown to have been covertly uploaded to the web server, encoded within the URL of an image on the page.

The example code to exploit this appears in Listing 6.1, producing an HTTP request as shown in Listing 6.2.

### 6.1.2 TLS — Protocol Verification

Once the HTTP-over-TLS (HTTPS) server was implemented, a variety of protocol interactions were tested. Initially standard HTTPS clients (wget, curl, Mozilla Firefox and Google Chrome) were used, and any issues encountered resolved; after this, the more comprehensive industry standard test suite SSL Labs from Qualys was employed, with results shown in Figure 6.3.

Listing 6.1: Floating point backdoor exploit

```
// Perform the rigged FDIV four times:
var a=1.91207592522993E+306/1.91207592522993E+306;
var b=1.91207592522993E+306/1.91207592522993E+306;
var c=1.91207592522993E+306/1.91207592522993E+306;
var d=1.91207592522993E+306/1.91207592522993E+306;
if (a!==1) {  // Result will be 1 on untampered CPU
  var k=a.toString(16)
   +','+b.toString(16)
   +','+c.toString(16)
   +','+d.toString(16);  // Concatenate the 4 values

  // Now put that key in the URL of the first image:
  document.images[0].src='?n='+k;
}
```

Listing 6.2: Resulting HTTP request to server

```
GET /floatback.php?n=efbeadde,77665544,bbaa9988,ffeeddcc
   HTTP/1.1
```

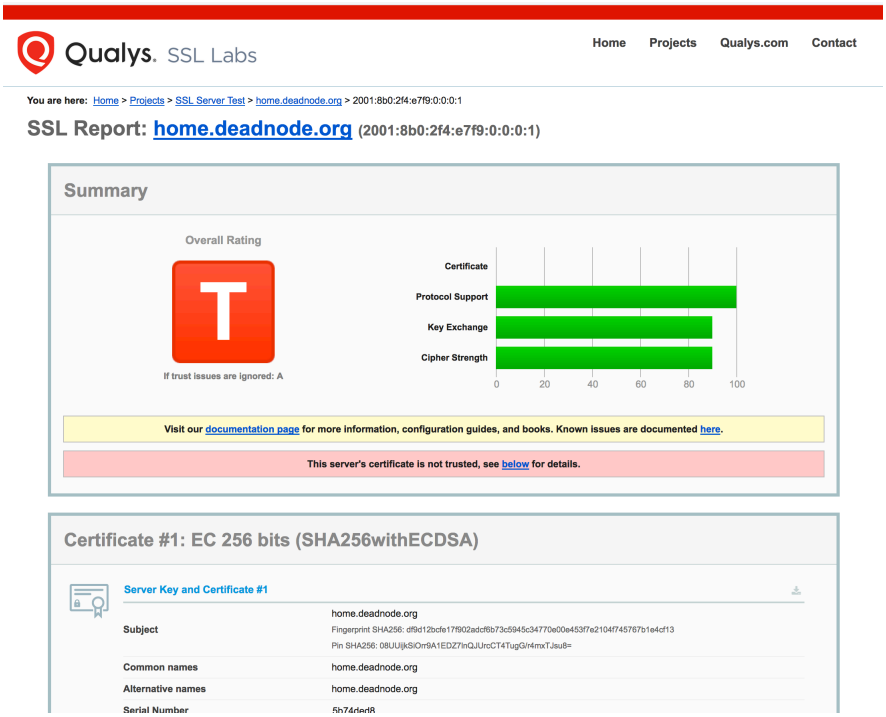



Figure 6.3: Qualys SSL Labs test results

The "T" score indicates a Trust issue — the test server is not configured with a publicly trusted certificate, issued by a genuine Certification Authority such as Verisign or LetsEncrypt — but all cryptographic and protocol aspects are correct; the test suite proceeds to simulate the cryptographic handshakes of a variety of common browsers. With the exception of Google Chrome on Windows XP Service Pack 3, which experiences a handshake failure, all compatible clients negotiate and connect correctly.

## 6.2 Micro-benchmarks

One important comparison is between the two full mode transitions (SMM and kernel mode) — particularly with the KAISER/KPTI enhanced isolation added

to the latter as mitigation to the Spectre/Meltdown exploits. (The `getpid` system call was chosen as the most trivial, since it only copies a non-sensitive constant integer; the `open` system call will be reading the file system cache, which is not readable from user mode, so incurs greater overhead in a full transition to restore access to kernel data.)

Since the secured server being developed in this project achieves the security benefits by transitioning into SMM before performing each signing operation, the relative performance impact of this change is indicated by the relationship between the 'signing' and 'SMM' figures: the signature operation in isolation takes a little less than the round-trip to and from SMM, 1.6 million processor ticks versus 2.4 million.

In normal usage `getpid` is faster than this, avoiding a system call entirely by returning the process's own copy of this value directly via a mechanism known as vDSO (Virtual Dynamic Shared Object).

The timing figures are shown in Table 6.1. Unfortunately the X200 system failed during testing, so further results could not be recorded; the remaining tests had to be performed on the fallback system alone, the T60.

SMI calls caused the unmodified T60 control laptop to freeze; this appears to be a known long-standing issue with the stock Lenovo BIOS (Ubuntu 2011).

The relative performance of the two hardware test platforms is indicated by comparing the first two columns (indicating the T60 has just under half the speed of the X200 on system calls), while comparing the two pairs of T60 figures ('T60' represents the test code running directly under Linux, 'T60 Qemu-VM' represents the same code executed under Qemu-VM simulation) indicates the relative performance penalty of the simulation system itself: approximately three orders of

magnitude slowdown (a factor of 1,000). On the most trivial system call, the additional overhead of simulation dominates (as shown by the much smaller difference between getpid and open/close times), but the relative performance of SMI invocation and open/close calls is more similar: 88 times slower in simulation versus 149 times slower on bare metal.

Statistics on the distribution of timings are shown in Table 6.2 and Table 6.3, obtained from running the output of the code in Appendix B through the Perl `Statistics::Descriptive::Full` library.

The maximum times for all operations are extreme outliers — around 3-5 million ticks on bare metal, around four times as high under KVM. Each indicates the test application was interrupted during that operation for something of the order of 2-20 ms.

The additional KVM overhead is most apparent when comparing the `getpid` operations (a median more than 17 times slower), closing to a factor of 7 for `open-close` and no discernable difference on cryptographic operations performed in userspace.

The SMI transition overhead is less uniform, with the upper quartile more than 55% higher than the lower — an interesting characteristic, worthy of further study elsewhere.

| Operation | X200 | T60 | | T60 Qemu-KVM | |
|---|---|---|---|---|---|
| Units | µs | µs | TSC | µs | TSC |
| NOP SMI | 448 | Not available | | 1310 | 2.4m |
| open/close | 3 | 7.1 | 3900 | 26 | 26k |
| getpid | 0.4 | 1.1 | 620 | 21 | 12k |
| Signing | Not available | 878 | 1.606m | 905 | 1.65m |

Table 6.1: Execution time for system calls and SMI invocations

| Operation | Minimum | 1st Quartile | Median | 3rd Quartile | Maximum |
|---|---|---|---|---|---|
| getpid | 1133 | 1155 | 1155 | 1155 | 5211503 |
| open-close | 6347 | 6479 | 6512 | 6545 | 3776872 |
| sign | 1534995 | 1542285.25 | 1544378 | 1547757.75 | 2924856 |

Table 6.2: Execution time (TSC ticks) on bare metal

| Operation | Minimum | 1st Quartile | Median | 3rd Quartile | Maximum |
|---|---|---|---|---|---|
| getpid | 20229 | 20295 | 20317 | 20361 | 33031357 |
| open-close | 44902 | 45397 | 45496 | 45595 | 29565196 |
| sign | 1536480 | 1543069 | 1546578 | 1596921 | 12533972 |
| SMI | 2235276 | 2326436.75 | 2921712.5 | 3618389 | 26339800 |

Table 6.3: Execution time (TSC ticks) under KVM

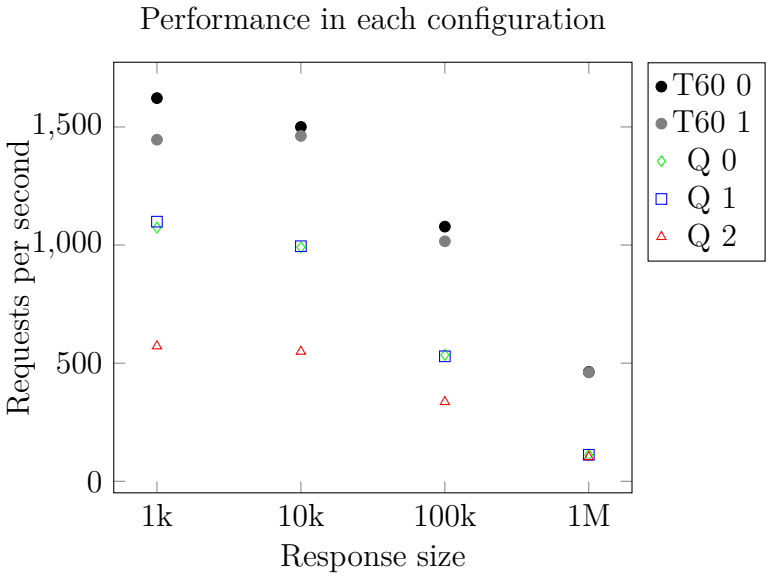
## 6.3 Macro-benchmarks

The performance recorded is shown in Figure 6.4.

The performance overhead of simulation as opposed to direct execution is apparent. Across the range of request sizes tested, physical hardware is consistently and proportionally faster than simulated. As the request size increases, the difference between SMM and other modes diminishes to less than 10% at the largest size, one MiB. Reasons for this, and the implications, are discussed in subsection 7.2.5.

## 6.4 Summary

Experiment 1 demonstrated the feasibility of a well concealed backdoor existing only on a hardware level, with no software modifications of any kind: a useful result in itself, confirming the importance of either being able to trust your CPU, or having robust precautions such as full airgap isolation in place, even after the sensitive information appears to have been discarded: small amounts of informa-




tion, such as the cryptographic keys being captured and then exfiltrated in this demonstration, can be covertly retained indefinitely for later retrieval.

Experiment 2 determines the mode transition costs for SMM entry and exit: non-trivial, but comparable to the time cost of the signing operation being protected, and substantially lower than for entry to Intel's SGX particularly when transferring any amount of data to the protected 'enclave' environment. (SMM can access shared data directly without performance penalty, while SGX's transition costs increase rapidly to 250 ms for relatively modest buffer sizes.) Experiment 2 also validates the use of SMM simulation for later tests and development work, since relative performance on non-trivial operations appears consistent within a factor of two of bare metal execution.

Experiments 3a-3c combine to demonstrate a working HTTPS server and establish an upper bound on the performance or latency cost of isolating the keys in two different ways, validating the original hypothesis about SMM's suitability for this technique. At the smallest extreme of payload sizes, where the cryptographic

handshake for each new connection dominates, the additional SMM overhead is of a similar magnitude; as the size increases, the impact of this extra overhead on overall throughput rapidly diminishes.

# 7

# Discussion

## 7.1 Introduction

This chapter analyses the experimental results described previously and describes their implication in the context of this research, along with discussion of potential further work building on this foundation.

## 7.2 Analysis of Results

### 7.2.1 Experiment 1: Backdoor

The floating-point backdoor experiment demonstrated a useful outcome in simulation: a hardware-only modification, exploitable from an unprivileged web page yet undetectable without specific prior knowledge of the backdoor's implementation.

### 7.2.2 Experiment 2: Microbenchmark

This experiment explored the performance impact of each transition round-trip between SMM and userspace code, compared to that of trivial and non-trivial system calls (which require an analogous transition to/from ring 0, kernel mode). On a kernel with Spectre/Meltdown mitigation in place as detailed in section 2.5, causing the memory mappings to be flushed on every transition, a non-trivial operation (open and close a cached file) is almost a full order of magnitude slower than a trivial one (`getpid`, which just copies the integer process PID from the kernel `task_struct`). Executing an SMI on the first test platform is two orders of magnitude slower still — 448 µs — but this is fast enough for over 2,000 transitions per second: sufficient to justify proceeding to stage 3 without further performance optimisation yet. (For comparison, Intel's SGX enclave is reported in (Gjerdrum et al. 2017) to take as much as 250 ms with half-megabyte buffers, almost three orders of magnitude slower than SMI entry, and substantially slower even without significant data transfer — on an Intel Core i5-6500 CPU at 3.20 GHz with four logical cores and 2×8 GB ofDDR3 DIMM DRAM, significantly faster hardware than used for these SMM tests.)

The simulation platform used for later testing was significantly slower than running on bare hardware, but for non-trivial operations the relative performance was still comparable.

Performance aspects and possible future optimisation opportunities are noted later in this chapter.

### 7.2.3 Experiment 3a: Protocol Handling

Once developed, the experimental server was validated with a variety of common HTTPS client programs (`curl`, `wget`, Mozilla Firefox and Google Chrome), then tested more thoroughly using the Qualys SSL Labs test, as noted in chapter 6. The server was operating without a publicly trusted certificate in place, but was otherwise fully compliant, attaining an 'A' score for the SSL/TLS implementation and completing all but one of the applicable simulated crytographic handshake sequences successfully. (That exception was an old version of Google Chrome from 2016 on Windows XP Service Pack 3; as Mozilla Firefox on the same platform and newer versions of Chrome on other platforms interoperate correctly, this issue did not seem worthy of closer examination for now.)

### 7.2.4 Experiment 3b: Process-isolated Key Handling

This verified the feasibility and performance impact of segregating the private key handling from other web server functionality, compared to the typical approach of handling keys directly within the main server. When the Heartbleed bug was discovered, allowing an attacker to read arbitary blocks of memory from within the target process, this approach led to many keys being compromised and forcing

urgent revocation and replacement.

By generating and holding the private key in a separate process, this would have been mitigated: some confidential information could still have been exposed, but the server keys at least would have remained safe — and the same isolation approach could be extended for other sensitive data such as credit card numbers or unencrypted passwords.

### 7.2.5 Experiment 3c: HTTPS Performance Testing

With a working HTTPS implementation using SMM security, full service testing and comparison against the non-SMM options gives the best indication of SMM's performance impact in the worst case. (As noted later, multiple options exist for reducing this impact if necessary.)

The relative performance on simulated hardware corroborates the micro-benchmark results: performing the cryptographic handshake computations in SMM approximately halves the rate at which handshakes are performed, causing a corresponding slowdown on the smallest requests (where this aspect dominates the overall server performance), falling to around 10% with 1 MiB requests.

The effect of size is to be expected: SSL/TLS uses two levels of encryption. First, the connection is established using public key cryptography. This handshake process negotiates two pairs of keys which are then used to encrypt and sign the data exchanged, and has a fixed computational cost regardless of the volume of data transferred later. Secondly, the request and response are encrypted and signed using those keys, taking time proportional to the volume involved. So, on small requests the former aspect dominates performance; on larger requests, the latter

becomes dominant.

The performance shown on the smallest requests, 572 1k requests per second, is also consistent with the bare metal SMM transition measurements from experiment 2 of 448 µs on a processor with approximately twice the performance (a higher clock speed and faster memory bus).

## 7.3 Hypothesis

The initial hypothesis was that SMM can be used to protect sensitive data such as cryptographic key material in a web server environment. The experimental results shown previously demonstrate that the approach is technically feasible, quantifying the performance impact — variable, determined by the size of objects being retrieved, from a worst case 50% slowdown on very small sizes, falling to 10% at 1 MiB.

## 7.4 Performance

### 7.4.1 Batch signing

The proof of concept server under test here performs a single cryptographic operation on each entry to SMM, then returns immediately. For a busy server, batching up operations and checking for additional queued requests before returning would allow the entry cost to be amortised across multiple requests, boosting throughput significantly.

For example, performing a batch of 10 signatures per transition could be expected to effectively reduce the transition overhead by 90% (at the expense of a

slight increase in latency to assemble each batch), rendering the overhead negligible (around 1%) for 1 MiB files.

### 7.4.2 Multiple cores/threads

In the standard Coreboot SMI handler, all cores receiving an SMI suspend execution until the handler routine completes execution on one core: the rest execute a `REP NOP` or `PAUSE` opcode (`f3 90`), which halts execution on that core until the next memory bus activity. The source code indicates this is done 'for security reasons' (Google 2013) without elaboration.

Allowing concurrent execution of SMM and non-SMM code may have security implications as implied by Google's Coreboot documentation, but multiple cores could be used to perform multiple crytographic operations in parallel to improve performance without creating additional issues.

## 7.5 Hardware alternatives

The primary alternative to this approach, where enhanced security is needed compared to direct key handling without extra isolation, is to use a dedicated cryptographic hardware device. Some PCs and servers are now equipped with a TPM (Trusted Platform Module), which provides a dedicated cryptographic and storage facility, with a fixed set of algorithms, limited storage and minimal performance (Bajikar 2002).

More advanced high-performance devices were commonly used for high-end services (Anderson et al. 2006) for a combination of performance and security reasons, but by 2007 software exploiting GPU acceleration reached comparable

performance to dedicated hardware (Manavski 2007), eroding this advantage even before mass-market processors were optimised for this.

For the most sensitive applications, an HSM still offers security benefits, particularly where performance is less critical — Certification Authorities use them to handle their long-validity signing keys, where the economic damage from any compromise would pose an existential threat to that company, the global DNS root, banks. Being specialist hardware, pricing is rarely public, but recent eBay listings offered two Thales devices for $3,999.00 and $5,999.00 for their nCipher nShield 500 and 6000 units respectively.

This work, then, occupies a middle ground between the high cost high security specialist devices and the more limited security protection of conventional software implementations; as noted in section 8.3, this approach could also be combined with the intrusion countermeasures of HyperCheck/HyperGuard in future to enhance this security.

## 7.6 Security

The focus of this research was on protection against various classes of memory attack such as buffer overflows, DMA attacks and cache manipulation.

On the experimental hardware platform used (and the majority of PCs) the BIOS is stored in a Flash memory chip which can be reprogrammed directly in software with only root privileges, which would allow an intruder with such access to replace or modify the SMM cryptographic code and thus replace the cryptographic keys with ones they could access. There are two well-established mitigations for this: either hardware write protection (for example, some Chromebook devices

have a screw which physically connects or disconnects a pin on the Flash chip to prevent unwanted modification regardless of access privileges); others allow write access to the BIOS only during a short window after reset, so updates are first downloaded to RAM, then the system is rebooted. At each boot, the existing BIOS checks for cryptographically signed updates waiting in RAM, and applies the update if present; this way, barring either compromise of the signing keys (Leyden 2017) or an exploit in the update mechanism, any intruder could only apply an authorised update to the BIOS, not their own arbitrary updates. Both mitigations could, of course, be combined if desired.

The test platform uses a relatively old processor design, predating Intel's addition of the `RDRAND` and `RDSEED` random number generation instructions. For the purposes of these experiments a simple pseudo-random number generator was used. For any real deployment a secure high quality random number generator should be substituted to avoid introducing weak or predictable keys; the special-purpose instructions are available within SMM and would provide sufficient entropy on a processor supporting them.

The attack surface of the cryptographic code running in SMM is intentionally minimal, performing no network access: any attacker, having already obtained privileged access to the target system, probing the SMM code would have access only to a signing operation, allowing them to provide a 256 bit value and request a signature of that. They could use this to obtain their own certificate corresponding to the protected private key (of course, any user connecting to the server also has access to download the certificate in use), and complete handshakes using that key as long as they maintained access to the compromised system — but this gives no persistence to the attacker, since the key itself remains protected.

## 7.7 Summary

The initial hypothesis from section 1.3 is confirmed by experiments demonstrating a functional web server using cryptographic keys secured through SMM isolation, and an indicative upper bound is established on the resulting performance penalty through multiple benchmarks.

# 8

# Summary, Conclusions

## 8.1 Introduction

The starting hypothesis from section 1.3 was:

> Secure isolation can be practically implemented using only the long-established Systems Management Mode mechanisms, giving better security isolation than existing techniques such as process separation.

The performance impact of SMM has been explored both on bare hardware and in virtualised form, and a proof of concept server demonstrated and benchmarked successfully.

Even on relatively old legacy hardware, with additional overhead, the performance impact due to SMM isolation was not prohibitive — approximately doubling the CPU time per handshake operation, causing a performance penalty falling from 50% on the smallest payload sizes (where the handshaking process dominates the overall workload) to 10% at 1 MiB. Opportunities for mitigating this performance loss further are also identified, with micro-benchmarking figures giving some indication what performance gain could be expected from this.

## 8.2 Dissemination

This work has already been depicted in a variety of conference poster presentations and internal events. A paper was prepared and submitted for the ESSoS conference in Germany, but that conference was cancelled at short notice; revision and resubmission of this paper for publication in an alternative venue is a promising course of action.

The two main pieces of source code used in this project are incorporated as appendices to this thesis; upon completion of the publication and review process, the full project code will be publicly released to enable easy re-use in future developments.

Some practical issues encountered during testing have already been shared with the LibreBoot team and are being addressed separately (in particular, the current build tooling is fragile, easily getting into a state in which compilation no longer

succeeds).

## 8.3 Further Work

This work has confirmed the potential for new uses of SMM in a security context, beyond that established by work such as (Wang, Stavrou and Ghosh 2010) on SMM protection of hypervisor and system integrity previously.

The experimentation also identified significant limitations in current SMM understanding and tools — since other research has focused so heavily on virtualisation, hypervisors and OS-level development, the lowest level firmware components have been largely neglected. Bugs in this area, when found in open source projects such as Qemu or production equipment such as the Thinkpad T60 used here, are often neglected and worked around by avoiding use of those features (for example, disabling the SMM-related power management functions on the T60 when encountered) rather than fixed, since these have been seen as low priority issues.

In recent months news such as Spectre and Meltdown have brought new attention to this area, researching low level hardware bugs and protection. Unlike reactive patching, SMM isolation provides proactive protection against issues of this type — hopefully this will bring new attention and resources to this previously overlooked area, improving security for us all.

### 8.3.1 Intrusion countermeasures

The HyperGuard/HyperCheck projects leveraged SMM as an integrity checking mechanism to detect and alert compromises of a system. These integrity checks could in future be incorporated within this work: not only would the keys in SMM

remain protected, the compromise would be detected and appropriate responses could be triggered, such as alerting the operator, clearing other data from the system as a precaution and shutting down or suspending the compromised machine to preserve evidence and prevent further exploitation.

### 8.3.2 Operation batching

As noted in subsection 7.4.1, significant gains in throughput are likely (in a server situation) from performing multiple cryptographic operations per transition to SMM and back: rather than passing individual requests immediately, combine the requests into sets and process a full set each time. This would amortise the transition cost across however many connection handshakes are being performed in that batch, trading increased throughput for increased latency determined by the batch size.

### 8.3.3 Multi core support

The SMI handler code in Coreboot pauses all cores on entry to SMM until the one active handler completes processing. Google's documentation indicates this is for 'security reasons' without elaboration — isolation between cores is a subject of active research at present, so avoiding concurrent execution entirely is a conservative response at the expense of some loss of throughput. With the implementation of batched request handling, effective use of multiple cores could be made without introducing new potential vulnerabilities since the code executing on each core would be at the same privilege level anyway. Processor core counts continue to increase, particularly in server environments, so the benefits here will

increase accordingly.

### 8.3.4 Additional algorithm support

This project's proof-of-concept code implemented only 256 bit elliptic curve signing, to demonstrate an application using SSL/TLS with ECDHA-SECP256R1-AES256-SHA256, sufficient for contemporary web browsers and benchmarking tools with HTTPS — generalisation to other key sizes and cryptographic algorithms would improve practical applications of this work in future.

### 8.3.5 Other applications and protocols

Particularly with the inclusion of other algorithms, the key protection and handling techniques demonstrated here could be applied to other protocols and applications such as SSH authentication, cryptocurrency transactions or a credential store akin to Microsoft's Credential Guard (which uses a special-purpose virtual machine to isolate credentials from the primary OS on desktop systems).

The two-level keying of DNSSEC (Eastlake 1999), in which each domain has a persistent Key Signing Key (KSK, to which the parent domain contains references) and a shorter duration Zone Signing Key (ZSK), could lend itself well to this system: each DNS server could generate a fresh ZSK within an SMM enclave on startup and export only the public key portion, with a more isolated (and hence less vulnerable to attack) system holding the KSK's private portion, needed only to sign a fresh ZSK when one of the DNS servers is installed or reset. This would allow DNS content to be updated more easily than the full offline signing currently used for sensitive domains with low update frequency.

### 8.3.6 Persistent storage

For the web server demonstrated in this project, creating a fresh keypair on initialisation is sufficient, but a mechanism for secure information persistence would be useful for other applications such as authentication. This could be a remote server (as in the case of credential caching, where a physically secured authentication server issues the transient tokens for client devices to retain and reuse, perhaps within a container such as Credential Guard), or a local component such as a or simply a non-volatile storage chip with access gated by the $\overline{\text{SMIACT}}$ control line, so only code executing within SMM can access it — similar to the mechanism already used to guard the system firmware against unauthorised updates.

### 8.3.7 SMM experimentation kit

Firmware experimentation still has practical barriers to overcome compared with application and operating system development. Early firmware inadvertently allowed the replacement of the SMI handler code by leaving the $D\_LOCK$ flag clear, enabling early SMM keyloggers and 'bootkits' to be installed without modifying the system firmware itself — a security issue, hence urgently patched, but also beneficial to experimenters. Restoring the legitimate uses of this without also restoring the security problems would be useful: perhaps a physical key or circuit jumper to permit experiments. Alternatively, a system which made use of SMM enclave code could be built with a cryptographic key, allowing signed code to be loaded (with appropriate verification) to configure or enhance functionality without replacing the system BIOS.

Open source projects such as Qemu, Bochs and CoreBoot/LibreBoot have

been invaluable in this work; commercial hardware developers such as Intel have high precision simulators of their hardware for internal development use, and the MPTLSim project (Zeng et al. 2009) led to the open-source MARSSx86 project (Patel et al. 2011), but this appears to have ceased development in 2012.

As noted previously in section 8.2, the public release of the software developed in this project is planned, hopefully in the form of a 'kit' to facilitate future firmware and SMM experimentation, making this much more accessible for future academic and open-source development.

# Appendices

# A

# API Design

The design of the SMI component's interface was driven by a preference for simplicity and a minimal attack surface, derived after study of some existing uses of SMI — in particular, a keylogger and the published information about simulating legacy (pre-USB) keyboards for operating systems lacking USB support such as MS DOS.

Required interactions:

- Generate keypair

| **Offset** (bytes) | **Size** (bytes) | **Content** |
|---|---|---|
| 0 | 64 | Public key (output) |
| 64 | 32 | SHA-256 hash to be signed (input) |
| 96 | 32 | Signature (output) |

Table A.1: Message passing area layout

- Export public key
- Sign message

For this project, a single 'mailbox' page (4096 bytes) of memory sufficed. The first two operations (generating a keypair and exporting the public key portion) are combined.

For TLS/SSL purposes, signing a message (which may be of variable length) consists of first hashing the message to produce a fixed-size message digest, then signing that digest value with the private key.

A page of memory is requested from the operating system, locked in memory with the `mlock` system call, then the underlying physical page address (determined by looking the virtual address up in `/proc/self/pagemap`) is communicated to the SMM handler. (Being 'below' the operating system, the SMM handler is not subject to the virtual address mappings.)

The interface, as implemented in the `do_smmcrypto` function in the code shown in Appendix F, requires loading a randomly chosen signature value ($0x9a69ec01$) into the ECX register and the page location into EBX before triggering an SMI.

For the purposes of this work, a single TLS algorithm combination was chosen: SHA-256 hashing with ECDSA on the SECP 256R1 elliptic curve, known as `ECDSA_SECP256R1_SHA256` — this is sufficient to provide compatibility with modern browsers for testing purposes, while the generalisation to different algorithms

and key sizes is clear but not directly beneficial at this stage.

When first called, a keypair is generated and written to the start of the communications page. The server code uses this key to generate an unsigned Certificate Signing Request, then calls the SMM handler a second time to sign this request. For public use, at this point the signed request would be sent to a Certification Authority, who would perform some checks and sign the request using their own private key to produce a publicly-trusted certificate chain; for experimental purposes, signing the certificate request using the same key again is sufficient, producing a 'self-signed' key.

For each new network connection, the signing function is invoked again, after writing the SHA256 hash of the message to be signed into the buffer. As noted in subsection 7.4.1, overall throughput could be improved in future (at the expense of some extra latency) by signing more than one message per invocation.

# B

# Micro benchmarking code

```
#define _GNU_SOURCE          /* See feature_test_macros(7)
    */
#include <sys/syscall.h>    /* For SYS_xxx definitions */
#include <sys/time.h>
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
```

```
#include <unistd.h>
#include <sys/io.h>
#include <sys/stat.h>
#include <fcntl.h>

#define uECC_CURVE uECC_secp256r1
#include "micro-ecc/uECC.h"

static uint64_t rdtsc(){
        unsigned int lo,hi;
        __asm__ __volatile__ ("rdtsc" : "=a" (lo), "=d" (
           hi));
        return ((uint64_t)hi << 32) | lo;
}

#ifndef BATCHONLY
static uint64_t *lapticks;

static void inittick(int n) {
        lapticks=malloc((n+1)*sizeof(uint64_t));
        if (lapticks==NULL) {
                fprintf(stderr,"Out of memory for tick
                   counter\n");
                abort();
```

```
        }
        lapticks[0]=rdtsc();
}

static void tick(int n) {
        lapticks[n+1]=rdtsc();
}

static void tickstop(const char *name,int n) {
        FILE *log=fopen(name,"a");
        if (!log) {
                perror("open log failed");
                free(lapticks);
                return;
        }
        for (int i=0;i<n;i++) {
                fprintf(log,"%lu\n",lapticks[i+1]-lapticks
                    [i]);
        }
        fclose(log);
        free(lapticks);
}
#endif
```

```
static int usage(const char *name) {
        fprintf(stderr,"Usage:\t%s <mode>\n",name);
        return 1;
}

static int sign(int iters) {
        uint8_t private_key[uECC_BYTES];
        uint8_t public_key[uECC_BYTES*2];
        // 32 random bytes standing in for SHA256 hash of
           payload for benchmarking
        uint8_t hash[]={0x6b,0x73,0x0c,0x63,0xe4,0x1b,0x17
           ,0x10,0x13,0x18,0xe9,0x94,0x89,0x75,0x4c,0xfe,0
           xde,0x29,0x7e,0xf7,0xfc,0xe1,0xef,0x62,0xc1,0
           x68,0x7c,0x05,0x52,0xf0,0x37,0x0a};
        uint8_t signature[uECC_BYTES];

        uECC_make_key(public_key,private_key);
#ifndef BATCHONLY
        inittick(iters);
#endif
        for (int i=0;i<iters;i++) {
                uECC_sign(private_key,hash,signature);
#ifndef BATCHONLY
                tick(i);
```

```
#endif
        }
        return 0;
}

static int st_syscall(int iters) {
        pid_t pid;
#ifndef BATCHONLY
        inittick(iters);
#endif
        for (int i=0;i<iters;i++) {
                pid=syscall(SYS_getpid);
#ifndef BATCHONLY
                tick(i);
#endif
        }
        return 0;
}

static int do_smi(int iters) {
        if (!ioperm(0xB2,1,1)) {
#ifndef BATCHONLY
        inittick(iters);
#endif
```

```
                for (int i=0;i<iters;i++) {
                        outb(1,0xb2);
#ifndef BATCHONLY
                tick(i);
#endif
                }
                return 0;
        }
        return 1;
}

static int st_open(int iters) {
        int fd;
#ifndef BATCHONLY
        inittick(iters);
#endif
        for (int i=0;i<iters;i++) {
                if ((fd=open("/etc/passwd",O_RDONLY))!=-1)
                    {
                        close(fd);
                }
#ifndef BATCHONLY
                tick(i);
#endif
```

```
        }
        return 0;
}

static int timer(char *name,struct timeval *tvs,uint64_t
    tickstart,int n,int rv) {
        struct timeval end;
        long int delta;
        if (!rv)
                tickstop(name,n);
        uint64_t tickend=rdtsc();
        gettimeofday(&end,NULL);
        delta=end.tv_sec-tvs->tv_sec;
        delta*=1000000;
        delta+=end.tv_usec-tvs->tv_usec;
        fprintf(stderr,"%s: %d iterations in %ld us/%ld
            ticks (= %ld ns / %ld ticks per call)\n",name,n
            ,delta,tickend-tickstart,1000*delta/n,(tickend-
            tickstart)/n);
        return rv;
}

int main(int argc,char **argv) {
        struct timeval start;
```

```
        uint64_t tickstart;
        if (argc!=2)
                return usage(*argv);
        gettimeofday(&start,NULL);
        tickstart=rdtsc();

        switch(*argv[1]) {
                case 'c':       // Cryptographic signing
                    operation
                return timer("sign",&start,tickstart,1000,
                    sign(1000));

                case 'o':       // open-close pair
#ifndef BATCHONLY
        inittick(1000000);
#endif
                return timer("open",&start,tickstart
                    ,1000000,st_open(1000000));

                case 's':       // NOP syscall
#ifndef BATCHONLY
        inittick(1000000);
#endif
                return timer("getpid",&start,tickstart
```

```
                    ,1000000,st_syscall(1000000));

                case 'i':       // NOP interrupt
#ifndef BATCHONLY
        inittick(1000);
#endif
                return timer("smi",&start,tickstart,1000,
                    do_smi(1000));
        }
        return 0;
}
```

# C

# C compiler

C compiler version information, as reported by '`gcc -V`'.

```
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/6/lto-
    wrapper
Target: x86_64-linux-gnu
```

```
Configured with: ../src/configure -v --with-pkgversion='
    Debian 6.3.0-18+deb9u1' --with-bugurl=file:///usr/share
    /doc/gcc-6/README.Bugs --enable-languages=c,ada,c++,
    java,go,d,fortran,objc,obj-c++ --prefix=/usr --program-
    suffix=-6 --program-prefix=x86_64-linux-gnu- --enable-
    shared --enable-linker-build-id --libexecdir=/usr/lib
    --without-included-gettext --enable-threads=posix --
    libdir=/usr/lib --enable-nls --with-sysroot=/ --enable-
    clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx
    -time=yes --with-default-libstdcxx-abi=new --enable-gnu
    -unique-object --disable-vtable-verify --enable-libmpx
    --enable-plugin --enable-default-pie --with-system-zlib
     --disable-browser-plugin --enable-java-awt=gtk --
    enable-gtk-cairo --with-java-home=/usr/lib/jvm/java
    -1.5.0-gcj-6-amd64/jre --enable-java-home --with-jvm-
    root-dir=/usr/lib/jvm/java-1.5.0-gcj-6-amd64 --with-jvm
    -jar-dir=/usr/lib/jvm-exports/java-1.5.0-gcj-6-amd64 --
    with-arch-directory=amd64 --with-ecj-jar=/usr/share/
    java/eclipse-ecj.jar --with-target-system-zlib --enable
    -objc-gc=auto --enable-multiarch --with-arch-32=i686 --
    with-abi=m64 --with-multilib-list=m32,m64,mx32 --enable
    -multilib --with-tune=generic --enable-checking=release
     --build=x86_64-linux-gnu --host=x86_64-linux-gnu --
    target=x86_64-linux-gnu
```

```
Thread model: posix
gcc version 6.3.0 20170516 (Debian 6.3.0-18+deb9u1)
```

# D

# Linux kernel

Linux kernel boot messages from the main test system used.

```
Linux version 4.9.0-8-amd64 (debian-kernel@lists.debian.
    org) (gcc version 6.3.0 20170516 (Debian 6.3.0-18+
    deb9u1) ) #1 SMP Debian 4.9.144-3.1 (2019-02-19)
Command line: BOOT_IMAGE=/boot/vmlinuz-4.9.0-8-amd64 root=
    UUID=68e52d36-4090-4d3e-95ae-eebd7f623a9e ro quiet
x86/fpu: Legacy x87 FPU detected.
```

```
e820: BIOS-provided physical RAM map:
BIOS-e820: [mem 0x0000000000000000-0x000000000009efff]
    usable
BIOS-e820: [mem 0x000000000009f000-0x000000000009ffff]
    reserved
BIOS-e820: [mem 0x00000000000d2000-0x00000000000d3fff]
    reserved
BIOS-e820: [mem 0x00000000000dc000-0x00000000000fffff]
    reserved
BIOS-e820: [mem 0x0000000000100000-0x00000000bf6cffff]
    usable
BIOS-e820: [mem 0x00000000bf6d0000-0x00000000bf6defff]
    ACPI data
BIOS-e820: [mem 0x00000000bf6df000-0x00000000bf6fffff]
    ACPI NVS
BIOS-e820: [mem 0x00000000bf700000-0x00000000bfffffff]
    reserved
BIOS-e820: [mem 0x00000000f0000000-0x00000000f3ffffff]
    reserved
BIOS-e820: [mem 0x00000000fec00000-0x00000000fec0ffff]
    reserved
BIOS-e820: [mem 0x00000000fed00000-0x00000000fed003ff]
    reserved
```

```
BIOS-e820: [mem 0x00000000fed14000-0x00000000fed19fff]
    reserved
BIOS-e820: [mem 0x00000000fed1c000-0x00000000fed8ffff]
    reserved
BIOS-e820: [mem 0x00000000fee00000-0x00000000fee00fff]
    reserved
BIOS-e820: [mem 0x00000000ff800000-0x00000000ffffffff]
    reserved
NX (Execute Disable) protection: active
SMBIOS 2.4 present.
DMI: LENOVO 1951CZ1/1951CZ1, BIOS 79ETE3WW (2.23 )
    09/12/2008
e820: update [mem 0x00000000-0x00000fff] usable ==>
    reserved
e820: remove [mem 0x000a0000-0x000fffff] usable
e820: last_pfn = 0xbf6d0 max_arch_pfn = 0x400000000
MTRR default type: uncachable
MTRR fixed ranges enabled:
  00000-9FFFF write-back
  A0000-BFFFF uncachable
  C0000-CFFFF write-protect
  D0000-DBFFF uncachable
  DC000-DFFFF write-back
  E0000-FFFFF write-protect
```

```
MTRR variable ranges enabled:
  0 base 000000000 mask F80000000 write-back
  1 base 080000000 mask FC0000000 write-back
  2 base 0BF700000 mask FFFF00000 uncachable
  3 base 0BF800000 mask FFF800000 uncachable
  4 disabled
  5 disabled
  6 disabled
  7 disabled
x86/PAT: Configuration [0-7]: WB  WC  UC- UC  WB  WC  UC-
   WT
found SMP MP-table at [mem 0x000f6810-0x000f681f] mapped
   at [ffff908d000f6810]
Base memory trampoline at [ffff908d00099000] 99000 size
   24576
BRK [0x89934000, 0x89934fff] PGTABLE
BRK [0x89935000, 0x89935fff] PGTABLE
BRK [0x89936000, 0x89936fff] PGTABLE
BRK [0x89937000, 0x89937fff] PGTABLE
BRK [0x89938000, 0x89938fff] PGTABLE
BRK [0x89939000, 0x89939fff] PGTABLE
BRK [0x8993a000, 0x8993afff] PGTABLE
RAMDISK: [mem 0x35cc5000-0x36e59fff]
ACPI: Early table checksum verification disabled
```

```
ACPI: RSDP 0x00000000000F67E0 000024 (v02 LENOVO)
ACPI: XSDT 0x00000000BF6D1322 00008C (v01 LENOVO TP-79
    00002230  LTP 00000000)
ACPI: FACP 0x00000000BF6D1400 0000F4 (v03 LENOVO TP-79
    00002230 LNVO 00000001)
ACPI BIOS Warning (bug): 32/64X length mismatch in FADT/
    Gpe0Block: 64/32 (20160831/tbfadt-603)
ACPI BIOS Warning (bug): Optional FADT field Gpe1Block has
     valid Address but zero Length: 0x000000000000102C/0x0
    (20160831/tbfadt-658)
ACPI: DSDT 0x00000000BF6D175E 00D467 (v01 LENOVO TP-79
    00002230 MSFT 0100000E)
ACPI: FACS 0x00000000BF6F4000 000040
ACPI: FACS 0x00000000BF6F4000 000040
ACPI: SSDT 0x00000000BF6D15B4 0001AA (v01 LENOVO TP-79
    00002230 MSFT 0100000E)
ACPI: ECDT 0x00000000BF6DEBC5 000052 (v01 LENOVO TP-79
    00002230 LNVO 00000001)
ACPI: TCPA 0x00000000BF6DEC17 000032 (v02 LENOVO TP-79
    00002230 LNVO 00000001)
ACPI: APIC 0x00000000BF6DEC49 000068 (v01 LENOVO TP-79
    00002230 LNVO 00000001)
ACPI: MCFG 0x00000000BF6DECB1 00003C (v01 LENOVO TP-79
    00002230 LNVO 00000001)
```

```
ACPI: HPET 0x00000000BF6DECED 000038 (v01 LENOVO TP-79
    00002230 LNVO 00000001)
ACPI: SLIC 0x00000000BF6DEE62 000176 (v01 LENOVO TP-79
    00002230  LTP 00000000)
ACPI: BOOT 0x00000000BF6DEFD8 000028 (v01 LENOVO TP-79
    00002230  LTP 00000001)
ACPI: SSDT 0x00000000BF6F2655 00025F (v01 LENOVO TP-79
    00002230 INTL 20050513)
ACPI: SSDT 0x00000000BF6F28B4 0000A6 (v01 LENOVO TP-79
    00002230 INTL 20050513)
ACPI: SSDT 0x00000000BF6F295A 0004F7 (v01 LENOVO TP-79
    00002230 INTL 20050513)
ACPI: SSDT 0x00000000BF6F2E51 0001D8 (v01 LENOVO TP-79
    00002230 INTL 20050513)
ACPI: Local APIC address 0xfee00000
No NUMA configuration found
Faking a node at [mem 0x0000000000000000-0
    x00000000bf6cffff]
NODE_DATA(0) allocated [mem 0xbf6cb000-0xbf6cffff]
Zone ranges:
  DMA      [mem 0x0000000000001000-0x0000000000ffffff]
  DMA32    [mem 0x0000000001000000-0x00000000bf6cffff]
  Normal   empty
  Device   empty
```

```
Movable zone start for each node
Early memory node ranges
  node   0: [mem 0x0000000000001000-0x000000000009efff]
  node   0: [mem 0x0000000000100000-0x00000000bf6cffff]
Initmem setup node 0 [mem 0x0000000000001000-0
    x00000000bf6cffff]
On node 0 totalpages: 783982
  DMA zone: 64 pages used for memmap
  DMA zone: 21 pages reserved
  DMA zone: 3998 pages, LIFO batch:0
  DMA32 zone: 12188 pages used for memmap
  DMA32 zone: 779984 pages, LIFO batch:31
Reserving Intel graphics memory at 0x00000000bf800000-0
    x00000000bfffffff
ACPI: PM-Timer IO Port: 0x1008
ACPI: Local APIC address 0xfee00000
ACPI: LAPIC_NMI (acpi_id[0x00] high edge lint[0x1])
ACPI: LAPIC_NMI (acpi_id[0x01] high edge lint[0x1])
IOAPIC[0]: apic_id 1, version 32, address 0xfec00000, GSI
    0-23
ACPI: INT_SRC_OVR (bus 0 bus_irq 0 global_irq 2 dfl dfl)
ACPI: INT_SRC_OVR (bus 0 bus_irq 9 global_irq 9 high level
    )
ACPI: IRQ0 used by override.
```

```
ACPI: IRQ9 used by override.
Using ACPI (MADT) for SMP configuration information
ACPI: HPET id: 0x8086a201 base: 0xfed00000
smpboot: Allowing 2 CPUs, 0 hotplug CPUs
PM: Registered nosave memory: [mem 0x00000000-0x00000fff]
PM: Registered nosave memory: [mem 0x0009f000-0x0009ffff]
PM: Registered nosave memory: [mem 0x000a0000-0x000d1fff]
PM: Registered nosave memory: [mem 0x000d2000-0x000d3fff]
PM: Registered nosave memory: [mem 0x000d4000-0x000dbfff]
PM: Registered nosave memory: [mem 0x000dc000-0x000fffff]
e820: [mem 0xc0000000-0xefffffff] available for PCI
   devices
Booting paravirtualized kernel on bare hardware
clocksource: refined-jiffies: mask: 0xffffffff max_cycles:
    0xffffffff, max_idle_ns: 7645519600211568 ns
setup_percpu: NR_CPUS:512 nr_cpumask_bits:512 nr_cpu_ids:2
    nr_node_ids:1
percpu: Embedded 35 pages/cpu @ffff908dbf400000 s105304
   r8192 d29864 u1048576
pcpu-alloc: s105304 r8192 d29864 u1048576 alloc=1*2097152
pcpu-alloc: [0] 0 1
Built 1 zonelists in Node order, mobility grouping on.
   Total pages: 771709
Policy zone: DMA32
```

```
Kernel command line: BOOT_IMAGE=/boot/vmlinuz-4.9.0-8-
    amd64 root=UUID=68e52d36-4090-4d3e-95ae-eebd7f623a9e ro
     quiet
PID hash table entries: 4096 (order: 3, 32768 bytes)
Calgary: detecting Calgary via BIOS EBDA area
Calgary: Unable to locate Rio Grande table in EBDA -
    bailing!
Memory: 3052708K/3135928K available (6268K kernel code,
    1161K rwdata, 2872K rodata, 1424K init, 656K bss, 83220
    K reserved, 0K cma-reserved)
Kernel/User page tables isolation: enabled
Hierarchical RCU implementation.
        Build-time adjustment of leaf fanout to 64.
        RCU restricting CPUs from NR_CPUS=512 to
            nr_cpu_ids=2.
RCU: Adjusting geometry for rcu_fanout_leaf=64, nr_cpu_ids
    =2
NR_IRQS:33024 nr_irqs:440 16
Console: colour VGA+ 80x25
console [tty0] enabled
clocksource: hpet: mask: 0xffffffff max_cycles: 0xffffffff
    , max_idle_ns: 133484882848 ns
hpet clockevent registered
tsc: Fast TSC calibration using PIT
```

```
tsc: Detected 1828.235 MHz processor
```

# E

# Processor

Processor information, as reported by the Linux kernel in `/proc/cpuinfo`.

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 15
model name      : Intel(R) Core(TM)2 CPU          T5600  @
    1.83GHz
```

```
stepping        : 2
microcode       : 0x57
cpu MHz         : 1000.000
cache size      : 2048 KB
physical id     : 0
siblings        : 2
core id         : 0
cpu cores       : 2
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 10
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic
   sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx
   fxsr sse sse2 ss ht tm pbe syscall nx lm constant_tsc
   arch_perfmon pebs bts rep_good nopl aperfmperf pni
   dtes64 monitor ds_cpl vmx est tm2 ssse3 cx16 xtpr pdcm
   lahf_lm kaiser tpr_shadow dtherm
bugs            : cpu_meltdown spectre_v1 spectre_v2
   spec_store_bypass l1tf
bogomips        : 3656.47
clflush size    : 64
```

```
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
power management:

processor       : 1
vendor_id       : GenuineIntel
cpu family      : 6
model           : 15
model name      : Intel(R) Core(TM)2 CPU          T5600  @
   1.83GHz
stepping        : 2
microcode       : 0x57
cpu MHz         : 1000.000
cache size      : 2048 KB
physical id     : 0
siblings        : 2
core id         : 1
cpu cores       : 2
apicid          : 1
initial apicid  : 1
fpu             : yes
fpu_exception   : yes
cpuid level     : 10
wp              : yes
```

```
flags           : fpu vme de pse tsc msr pae mce cx8 apic
   sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx
   fxsr sse sse2 ss ht tm pbe syscall nx lm constant_tsc
   arch_perfmon pebs bts rep_good nopl aperfmperf pni
   dtes64 monitor ds_cpl vmx est tm2 ssse3 cx16 xtpr pdcm
   lahf_lm kaiser tpr_shadow dtherm
bugs            : cpu_meltdown spectre_v1 spectre_v2
   spec_store_bypass l1tf
bogomips        : 3656.47
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
power management:
```

# F

# Multi-mode HTTPS server code

```
/*
    Original TLS server example: https://wiki.openssl.org/
        index.php/Simple_TLS_Server
    Certificate handling: https://stackoverflow.com/questions
        /16364522/how-do-i-create-a-self-signed-certificate-in
        -openssl-programatically-i-e-not
```

```
Main JS changes:
Sign with the key BEFORE discarding it (yes, RLY)
Stub out pthreads - not used, avoids SIGILL problem early
    in dev
Enable SO_REUSEADDR

Elliptic curve info from:
http://fm4dd.com/openssl/eckeycreate.htm

CSR code from:
http://openssl.6102.n7.nabble.com/create-certificate-
    request-programmatically-using-OpenSSL-API-td29197.html
*/

#define PAGE_SHIFT 12

#ifdef __GNUC__
#  define UNUSED(x) UNUSED_ ## x __attribute__((__unused__
    ))
#else
#  define UNUSED(x) UNUSED_ ## x
#endif

#include <stddef.h>
```

```
#include "micro-ecc/uECC.h"
#include "sha-256.h"

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <openssl/ssl.h>
#include <openssl/err.h>
#include <openssl/x509v3.h>
#include <sys/mman.h>
#ifdef __linux__
#include <malloc.h>
#include <sys/types.h>
#endif
#include <sys/stat.h>
#include <fcntl.h>


// JS prototype for OPENSSL_cpuid_setup and
   x509_digest_sign_algorithm
void OPENSSL_cpuid_setup(void);
```

```
int x509_digest_sign_algorithm(EVP_MD_CTX *ctx, X509_ALGOR
    *algor);

X509 *owncert;

uint8_t private_key[uECC_BYTES];
uint8_t public_key[uECC_BYTES*2];
EC_KEY *pubkey;
#ifdef USE_SMM
void *mailslot=NULL;
#if USE_SMM == 1
int ipcpipes[2][2];
#endif
#if USE_SMM == 2
static uint8_t *commpage=0;
uint32_t pagenum;
#endif
#endif

int siglen(uint8_t *out,uint8_t sig[uECC_BYTES*2]) {
  int len=0;
  out[0]=0x30;
  out[2]=2;
  len=3;
```

```
    out[len++]=uECC_BYTES;
    if (sig[0]>127) {
      out[len-1]++;
      out[len++]=0; // Zero pad for signing hack
    }
    memcpy(out+len,sig,uECC_BYTES);
    len+=uECC_BYTES;
    out[len++]=2;
    out[len++]=uECC_BYTES;
    if (sig[uECC_BYTES]>127) {
      out[len-1]++;
      out[len++]=0; // Zero pad for signing hack
    }
    memcpy(out+len,sig+uECC_BYTES,uECC_BYTES);
    len+=uECC_BYTES;
    out[1]=len-2;
    return len;
}

#ifdef USE_SMM

#if USE_SMM == 1
    int worker_r,worker_w;
    uint8_t worker_c;
```

```
void keyworker(int r,int w) {
  uint8_t c='\0';
  uECC_make_key(public_key,private_key);
  memcpy(mailslot,public_key,uECC_BYTES*2);
  if (write(w,&c,sizeof(c))==-1) {
    perror("write IPC init");
    return;
  }
  while(1) {
    switch(read(r,&c,sizeof(c))) {
      case -1:
      perror("read");
      // Fall through
      case 0: // EOF: shutdown gracefully
      return;

      default: // Work to do! Sign the message in
          mailslot
      uECC_sign(private_key,mailslot,mailslot+32);
      if (write(w,&c,sizeof(c))==-1) {
        perror("write IPC");
        return;
      }
```

```
        }
      }
    }
#endif

#if USE_SMM == 2
    int do_smmcrypto(void) {
      int result;
      /* Call SMI by writing 0x01 to I/O 0xb2 */
      __asm__ volatile (
        "movl $0x9a69ec01,%%ecx\n\t"
        "movl %1,%%ebx\n\t"
        "outb %%al,$0xb2\n\t"
        "movl %%ebx,%0"
      :"=r"(result)
      :"r"(pagenum)
          :"ecx","ebx","memory");
      fprintf(stderr,"SMM returned %08x\n",result);
      return result;
    }
#endif


enum ssl_private_key_result_t smm_sign(
```

```
    SSL * UNUSED(ssl),
    uint8_t *out,
    size_t *out_len,
    size_t max_out,
    uint16_t signature_algorithm,
    const uint8_t *in,
    size_t in_len
    ) {
#if USE_SMM == 0
    uint8_t hash[32], sig[uECC_BYTES*2];
#endif
    switch(signature_algorithm) {
      case SSL_SIGN_ECDSA_SECP256R1_SHA256:
      fprintf(stderr,"sign(%lu bytes, alg %d)\n",max_out,
          signature_algorithm);
#if USE_SMM == 0
      calc_sha_256(hash,in,in_len);
      uECC_sign(private_key,hash,sig);
      *out_len=siglen(out,sig);
#elif USE_SMM == 1 // Use worker process
      calc_sha_256(mailslot,in,in_len);
      if (write(worker_w,&worker_c,sizeof(worker_c))!=sizeof
          (worker_c)) {
        perror("Worker IPC error");
```

```
        return ssl_private_key_failure;
      }
      if (read(worker_r,&worker_c,sizeof(worker_c))!=sizeof(
          worker_c)) {
        perror("Worker IPC error");
        return ssl_private_key_failure;
      }
      *out_len=siglen(out,mailslot+32);
#elif USE_SMM == 2
      calc_sha_256(commpage+(uECC_BYTES*2),in,in_len);
      do_smmcrypto(); // SMI magic puts signature in
          commpage now
      *out_len=siglen(out,commpage+(uECC_BYTES*2)+32);
#endif
      return ssl_private_key_success;
      break;

      default:
      fprintf(stderr,"Unknown signature algorithm requested:
          0x%04x\n",signature_algorithm);
      return ssl_private_key_failure;
    }
}
enum ssl_private_key_result_t smm_decrypt(SSL * UNUSED(ssl
```

```
    ), uint8_t * UNUSED(out),
size_t * UNUSED(out_len), size_t max_out,
const uint8_t * UNUSED(in), size_t in_len) {
  fprintf(stderr,"decrypt(%lu in,%lu out)\n",in_len,
     max_out);
  return ssl_private_key_failure;
}
enum ssl_private_key_result_t smm_complete(SSL * UNUSED(
    ssl), uint8_t * UNUSED(out),
size_t * UNUSED(out_len), size_t UNUSED(max_out)) {
  return ssl_private_key_failure;
}
const SSL_PRIVATE_KEY_METHOD smm_method = {
  smm_sign,
  smm_decrypt,
  smm_complete
};
#endif


int create_socket(int port)
{
    int s,reuse=1;
    struct sockaddr_in addr;
```

```
    addr.sin_family = AF_INET;
    addr.sin_port = htons(port);
    addr.sin_addr.s_addr = htonl(INADDR_ANY);

    s = socket(AF_INET, SOCK_STREAM, 0);
    if (s < 0) {
  perror("Unable to create socket");
  exit(EXIT_FAILURE);
    }

    if (setsockopt(s,SOL_SOCKET,SO_REUSEADDR,&reuse,sizeof
        (reuse))) {
      perror("setsockopt(SO_REUSEADDR)");
      exit(EXIT_FAILURE);
    }

    if (bind(s, (struct sockaddr*)&addr, sizeof(addr)) <
        0) {
  perror("Unable to bind");
  exit(EXIT_FAILURE);
    }

    if (listen(s, 1) < 0) {
```

```
    perror("Unable to listen");
    exit(EXIT_FAILURE);
    }

    return s;
}

void init_openssl()
{
    SSL_load_error_strings();
    OpenSSL_add_ssl_algorithms();
}

void cleanup_openssl()
{
    EVP_cleanup();
}

SSL_CTX *create_context()
{
    const SSL_METHOD *method;
    SSL_CTX *ctx;

    method = SSLv23_server_method();
```

```
    ctx = SSL_CTX_new(method);
    if (!ctx) {
      perror("Unable to create SSL context");
      ERR_print_errors_fp(stderr);
      exit(EXIT_FAILURE);
    }

#ifdef USE_SMM
    SSL_CTX_set_private_key_method(ctx,&smm_method);
#endif
    return ctx;
}

EVP_PKEY *PrivateKey = NULL;

EVP_PKEY *newerkey() {
#ifdef USE_SMM
    uint8_t octetbuff[1+uECC_BYTES*2]={0x4};
#endif
    EVP_PKEY *privkey=EVP_PKEY_new();
    EC_KEY *k;
    int grp=OBJ_txt2nid("prime256v1");
    //int grp=OBJ_txt2nid("brainpoolP256r1");
```

```
  k=EC_KEY_new_by_curve_name(grp);

#ifdef USE_SMM
  // JS SMM bits later 2018-08-06:
  const EC_GROUP *pubkeygroup=EC_KEY_get0_group(k);
  EC_POINT * pubkeypoint=EC_POINT_new(pubkeygroup);
  memcpy(octetbuff+1,public_key,uECC_BYTES*2);
  EC_POINT_oct2point(pubkeygroup,pubkeypoint,octetbuff,1+
      uECC_BYTES*2,NULL);
#endif

  if (!k) {
    fprintf(stderr,"ECC init failed for group %d with
        error %s\n",grp,ERR_error_string(ERR_get_error(),
        NULL));
    return NULL;
  }
  EC_KEY_set_asn1_flag(k,OPENSSL_EC_NAMED_CURVE);
#ifdef USE_SMM
  EC_KEY_set_public_key(k,pubkeypoint);
//  X509_PUBKEY_set0_param(pub,OBJ_nid2obj(
    NID_X9_62_id_ecPublicKey))

#else
```

```
    if (!EC_KEY_generate_key(k)) {
      fprintf(stderr,"ECC keygen failed\n");
      return NULL;
    }
#endif
    if (!EVP_PKEY_assign_EC_KEY(privkey,k)) {
      fprintf(stderr,"ECC conversion failed\n");
      return NULL;
    }
    return privkey;
}

EVP_PKEY *newkey() {
    RSA *KeyPair;
    BIGNUM *BigNumber = NULL;

    // Create the RSA key pair object
    KeyPair = RSA_new();
    if (!KeyPair)
      return NULL;

    // Create the big number object
    BigNumber = BN_new();
    if (!BigNumber)
```

```
    return NULL;

  // Set the word
  if (!BN_set_word (BigNumber, 65537))
    return NULL;

  // Generate the key pair; lots of computes here
  if (!RSA_generate_key_ex (KeyPair, 4096, BigNumber, NULL
     ))
    return NULL;

  // Now we need a private key object
  PrivateKey = EVP_PKEY_new();
  if (!PrivateKey)
    return NULL;

  // Assign the key pair to the private key object
  if (!EVP_PKEY_assign_RSA (PrivateKey, KeyPair))
    return NULL;

  // KeyPair now belongs to PrivateKey, so don't clean it
     up separately
  KeyPair = NULL;
  return PrivateKey;
```

```
}

int signcert(X509 *Cert) {
    EVP_MD_CTX ctx;
   void *asn=Cert->cert_info;
   ASN1_BIT_STRING *signature=Cert->signature;
   X509_ALGOR *algor1=Cert->cert_info->signature,*algor2=
      Cert->sig_alg;
   const ASN1_ITEM *it=ASN1_ITEM_rptr(X509_CINF);

    EVP_MD_CTX_init(&ctx);

   Cert->cert_info->enc.modified = 1;
   // Sign it with SHA-256 - inlined variant of X509_sign
      for now
    if (!EVP_DigestSignInit(&ctx, NULL, EVP_sha256(), NULL,
        PrivateKey)) {
        EVP_MD_CTX_cleanup(&ctx);
        return 0;
    }

    unsigned char *buf_in = NULL, *buf_out = NULL;
    size_t inl = 0, outl = 0;
```

```
    /* Write out the requested copies of the
       AlgorithmIdentifier. */
    if (algor1 && !x509_digest_sign_algorithm(&ctx, algor1)
       ) {
        goto err;
    }
    if (algor2 && !x509_digest_sign_algorithm(&ctx, algor2)
       ) {
        goto err;
    }

    inl = ASN1_item_i2d(asn, &buf_in, it);

    outl = EVP_PKEY_size(PrivateKey);
    buf_out = OPENSSL_malloc((unsigned int)outl);
    if ((buf_in == NULL) || (buf_out == NULL)) {
        outl = 0;
        OPENSSL_PUT_ERROR(X509, ERR_R_MALLOC_FAILURE);
        goto err;
    }

#ifdef USE_SMM
  smm_sign(NULL, buf_out,&outl, outl,
    SSL_SIGN_ECDSA_SECP256R1_SHA256, buf_in, inl);
```

```
#else
    if (!EVP_DigestSign(&ctx, buf_out, &outl, buf_in, inl))
        {
        outl = 0;
        OPENSSL_PUT_ERROR(X509, ERR_R_EVP_LIB);
        goto err;
    }
#endif
    if (signature->data != NULL)
        OPENSSL_free(signature->data);
    signature->data = buf_out;
    buf_out = NULL;
    signature->length = outl;
    /*
     * In the interests of compatibility, I'll make sure
        that the bit string
     * has a 'not-used bits' value of 0
     */
    signature->flags &= ~(ASN1_STRING_FLAG_BITS_LEFT | 0x07
       );
    signature->flags |= ASN1_STRING_FLAG_BITS_LEFT;
err:
    EVP_MD_CTX_cleanup(&ctx);
    OPENSSL_free(buf_in);
```

```
    OPENSSL_free(buf_out);
    return (outl);
}

const unsigned char certissuance[]="Certificate Issuance";
X509 *CreateCertificate (const unsigned char *Country,
    const unsigned char *State, const unsigned char *
    Locality, const unsigned char *OrganizationName, const
    unsigned char *CommonName, const unsigned char *DNSName
    , int Serial, int DaysValid)
{
  X509 *Cert = NULL;
  X509_NAME *Name = NULL;
  RSA *KeyPair = NULL;
  BIGNUM *BigNumber = NULL;
  int Success = 0;



  // Faux loop...
  do {
    // Create the certificate object
    Cert = X509_new();
    if (!Cert)
```

```
    break;

// Set version 2, and get version 3
X509_set_version (Cert, 2);

// Set the certificate's properties
ASN1_INTEGER_set (X509_get_serialNumber (Cert), Serial
    );
X509_gmtime_adj (X509_get_notBefore (Cert), 0);
X509_gmtime_adj (X509_get_notAfter (Cert), (long)(60 *
    60 * 24 * (DaysValid ? DaysValid : 1)));
Name = X509_get_subject_name (Cert);
if (Country && *Country)
  X509_NAME_add_entry_by_txt (Name, "C", MBSTRING_ASC,
      Country, -1, -1, 0);

// JS 2018-02-05 Add (S)tate and (L)ocality fields to
    get valid CSR
if (State && *State)
  X509_NAME_add_entry_by_txt (Name, "ST", MBSTRING_ASC
      , State, -1, -1, 0);
if (Locality && *Locality)
  X509_NAME_add_entry_by_txt (Name, "L", MBSTRING_ASC,
      Locality, -1, -1, 0);
```

```
X509_NAME_add_entry_by_txt (Name, "OU", MBSTRING_ASC,
    certissuance, -1, -1, 0);
if (OrganizationName && *OrganizationName)
  X509_NAME_add_entry_by_txt (Name, "O", MBSTRING_ASC,
      OrganizationName, -1, -1, 0);

if (CommonName && *CommonName)
  X509_NAME_add_entry_by_txt (Name, "CN", MBSTRING_ASC
      , CommonName, -1, -1, 0);
X509_set_issuer_name (Cert, Name);

// Set the DNS name
if (DNSName && *DNSName)
{  X509_EXTENSION *Extension;
  char Buffer[512];

  // Format the value
  sprintf (Buffer, "DNS:%s", DNSName);
  Extension = X509V3_EXT_conf_nid (NULL, NULL,
     NID_subject_alt_name, Buffer);
  if (Extension)
  {
    X509_add_ext (Cert, Extension, -1);
    X509_EXTENSION_free (Extension);
```

```
    }
    else {
      fprintf(stderr,"SAN extension failed!\n");
    }
  }

  // JS key handling
  PrivateKey=newerkey();
  fprintf(stderr,"Key created at %p\n",PrivateKey);

  // Set the certificate's public key from the private
     key object
  if (!X509_set_pubkey (Cert, PrivateKey))
    break;

  if (!signcert(Cert))
    break;

  // PrivateKey now belongs to Cert, so don't clean it
     up separately
  // Actually reused later! PrivateKey = NULL;

  // Success
  fprintf(stderr,"Cert created OK\n");
```

```
    Success = 1;

  } while (0);

  // Things we always clean up
  if (BigNumber)
    BN_free (BigNumber);

  // Things we clean up only on failure
  if (!Success)
  {
    fprintf(stderr,"Certificate init error:\n");
    ERR_print_errors_fp(stderr);
    if (Cert)
      X509_free (Cert);
    if (PrivateKey)
      EVP_PKEY_free (PrivateKey);
    if (KeyPair)
      RSA_free (KeyPair);
    Cert = NULL;
  }

  // Return the certificate (or NULL)
  return (Cert);
```

```
}


void configure_context(SSL_CTX *ctx)
{
  SSL_CTX_set_ecdh_auto(ctx, 1);

  /* Set the key and cert */
  if (SSL_CTX_use_certificate(ctx,owncert) <= 0) {
    ERR_print_errors_fp(stderr);
    exit(EXIT_FAILURE);
  }

  if (SSL_CTX_use_PrivateKey(ctx, PrivateKey) <= 0 ) {
    ERR_print_errors_fp(stderr);
    exit(EXIT_FAILURE);
  }


  // JS 2018-02-05 Use decent ciphers only!
  SSL_CTX_set_cipher_list(ctx,"EECDH+AESGCM:EDH+AESGCM:
      AES256+EECDH:AES256+EDH:ECDH+AES128:RSA+AES128");
}
```

```
void printcsr() {
  X509_REQ *req;

  if (!(req=X509_to_X509_REQ(owncert,PrivateKey,EVP_sha256
    ()))) {
    fprintf(stderr,"Failed to initialise CSR\n");
  } else {
    PEM_write_X509_REQ(stderr,req);
    X509_REQ_free(req);
  }
}

#ifdef USE_SMM
#if USE_SMM == 2
static int init_smm() {
  int mapfd;
  uint64_t pagedata;

  if ((commpage=memalign(1<<PAGE_SHIFT,1<<PAGE_SHIFT))) {
    memset(commpage,11,1<<PAGE_SHIFT);
    if (!mlock(commpage,1<<PAGE_SHIFT)) {
      if (!ioperm(0xB2,1,1)) {
        if ((mapfd=open("/proc/self/pagemap",O_RDONLY))
          ==-1) {
```

```
            perror("Could not access /proc/self/pagemap:");
        } else {
            if (pread(mapfd,&pagedata,sizeof(pagedata),((
                uint64_t)commpage)>>(PAGE_SHIFT-3))==sizeof(
                pagedata)) {
                printf("Page data: 0x%016lx\n",pagedata);
                if (pagedata & 1ULL<<63) { // Page present in
                    bits[0:54]
                    pagedata &= (1ULL<<55)-1; // Extract just
                        the page number bits
//                  pagedata<<=PAGE_SHIFT;
                    if (pagedata<(1ULL<<20)) {
                        pagenum=pagedata;
                        close(mapfd);
                        fprintf(stderr,"Mailslot page established,
                            virtual %p physical 0x%08x",commpage,
                            pagenum);
                        do_smmcrypto();
                        fprintf(stderr,"Public key retrieved, 
                            first octet is %02x\n",*commpage);
                        memcpy(public_key,commpage,uECC_BYTES*2);
                        return 1;
                    } else {
                        fprintf(stderr,"Workspace page %lx outside
```

```
                        bottom 4 Gb\n",pagedata); // 714000
                    }
                }
            }
            close(mapfd);
        }
    }
    munlock(commpage,1<<PAGE_SHIFT);
  }
  free(commpage);
  commpage=0;
 }
 return 0;
}
#endif
#endif

const unsigned char
  country[]="GB",
  state[]="Scotland",
  city[]="Perth",
  orgname[]="James Sutherland",
  commonname[]="home.deadnode.org";
```

```
int main(int UNUSED(argc), char ** UNUSED(argv))
{
  struct timeval tv;
  int sock;
  SSL_CTX *ctx;

#ifdef USE_SMM
  mailslot=mmap(NULL,4096,PROT_READ|PROT_WRITE,
    MAP_ANONYMOUS|MAP_SHARED,-1,0);
  if (mailslot==MAP_FAILED) {
    perror("mmap() failed!\n");
    return -1;
  }
#if USE_SMM == 0 // Inline mode
  uECC_make_key(public_key,private_key);
#elif USE_SMM == 1 // Forked mode
  if (pipe(ipcpipes[0])==-1 || pipe(ipcpipes[1])==-1) {
    perror("pipe");
    return -1;
  }
  switch(fork()) {
    case -1:
    perror("fork");
    return -1;
```

```
      case 0: // Become worker process
      close(ipcpipes[0][1]);
      close(ipcpipes[1][0]);
      keyworker(ipcpipes[0][0],ipcpipes[1][1]);
      return 0;

      default:
      close(ipcpipes[0][0]);
      close(ipcpipes[1][1]);
      worker_r=ipcpipes[1][0];
      worker_w=ipcpipes[0][1];
      if (read(worker_r,&worker_c,sizeof(worker_c))!=sizeof(
          worker_c)) {
        perror("Worker init");
        return -1;
      }
      memcpy(public_key,mailslot,uECC_BYTES*2);
      break;
    }
#else // True SMM!
    if (!init_smm()) {
      return -1;
    }
```

```
#endif
#endif

    OPENSSL_cpuid_setup();
    init_openssl();

    gettimeofday(&tv,NULL);
    owncert=CreateCertificate(country,state,city,orgname,
        commonname,commonname,tv.tv_sec,365);
    printcsr();

    ctx = create_context();

    configure_context(ctx);

    sock = create_socket(4433);

    /* Handle connections */
    while(1) {
      struct sockaddr_in addr;
      uint len = sizeof(addr);
      SSL *ssl;
      const char reply[] = "HTTP/1.0 200 OK\r\nContent-Type:
          text/plain\r\n\r\n";
```

```
int client = accept(sock, (struct sockaddr*)&addr, &
    len);
if (client < 0) {
  perror("Unable to accept");
  exit(EXIT_FAILURE);
}

ssl = SSL_new(ctx);
SSL_set_fd(ssl, client);

if (SSL_accept(ssl) <= 0) {
  ERR_print_errors_fp(stderr);
}
else {
  char buff[1024];
  int n,fd;
  n=SSL_read(ssl,buff,sizeof(buff));
  if (n>0) {
    if ((fd=open("index.html",O_RDONLY))!=-1) {
      struct stat sbuff;
      if (fstat(fd,&sbuff)!=-1) {
        void *buff=malloc(sbuff.st_size);
        if (buff) {
```

```
            read(fd,buff,sbuff.st_size);
            SSL_write(ssl, reply, sizeof(reply));
            SSL_write(ssl, buff, sbuff.st_size);
            free(buff);
          } else {
            fprintf(stderr,"OOM!\n");
          }
        } else {
          perror("fstat(index.html)");
        }
        close(fd);
      } else {
        perror("index.html");
      }
    }
  }

  SSL_free(ssl);
  close(client);
}

close(sock);
SSL_CTX_free(ctx);
cleanup_openssl();
```

```
}
```

# Glossary

**ACME** Automatic Certificate Management Environment.

**AES** Advanced Encryption Standard.

**AESKEYGENASSIST** AES Key Generation Assist, the instruction used to expand raw AES encryption keys to the full schedule used for encryption.

**ARM** Advanced RISC Machines/Acorn RISC Machines.

**ARMORED** An implementation of the TRESOR encryption on ARM platforms.

**ASLR** Address Space Layout Randomisation.

**BIOS** Basic Input/Output System, the system firmware on the IBM PC.

**CA** Certificate Authority.

**CATT** CAn't Touch This, a RowHammer attack mitigation.

**CDN** Content Delivery Network.

**CLI** CLear Interrupt flag x86 instruction.

**CPU** Central Processing Unit.

**CPY** CoPY instruction.

**CS** Code Segment/Code Selector register on x86.

**CSR** Certificate Signing Request.

**CVE** Common Vulnerabilities and Exposures.

**DCI** Direct Connect Interface, an Intel debugging extension to USB 3.

**DDoS** Distributed Denial of Service attack.

**DMA** Direct Memory Access.

**DNS** Domain Name System.

**DOS** Disk Operating System.

**DoS** Denial of Service attack.

**DRAM** Dynamic Random Access Memory.

**FDIV** Floating Point Division x86 instruction.

**FPGA** Field Programmable Gate Array.

**FROST** Forensic Recovery Of Scrambled Telephones.

**GMU** George Mason University, Virginia, USA.

**HSM** Hardware Security Module.

**HTML** HyperText Markup Language.

**HTTP** HyperText Transfer Protocol, RFC 2616 and subsequent.

**HTTPS** HTTP Secure.

**IBM** International Business Machines.

**ICE** In Circuit Emulation.

**ICEBP** ICE BreakPoint.

**IEEE** Institute of Electrical and Electronics Engineers.

**IN** INput instruction.

**IOMMU** I/O Memory Management Unit.

**IOPB** I/O Permissions Bitmap.

**IOPL** I/O Privilege Level.

**IOV** I/O Virtualisation.

**JIT** Just In Time.

**JTAG** Joint Test Action Group.

**KAISER** Kernel And user Space Isolation.

**KPTI** Kernel Page Table Isolation.

**LOADALL** Load All register contents x86 instruction.

**MCH** Memory Controller Hub.

**MIPS** Microprocessor without Interlocked Pipeline Stages, also MIPS Technologies.

**MIT** Massachussetts Institute of Technology.

**MTRR** Memory Type Range Register.

**OS** Operating System.

**OUT** OUTput.

**OUTB** OUTput Byte x86 instruction.

**PCI** Peripheral Component Interconnect.

**RAM** Random Access Memory.

**RDS** ReaD Select.

**RSM** ReSuMe x86 instruction for returning from SMM.

**SGI** Silicon Graphics Inc.

**SGX** Software Guard Extensions.

**SMBASE** SMRAM BASE address register.

**SMEP** Supervisor Mode Execution Prevention.

**SMI** System Management Interrupt.

**SMIACT** SMI Active.

**SMM** System Management Mode.

**SMRAM** System Management RAM.

**SMRAMC** SMRAM Control register.

**SNI** Server Name Indication, RFC 3546, Transport Layer Security (TLS) Extensions.

**SPE** Synergistic Processing Element.

**SSL** Secure Sockets Layer.

**STI** SeT Interrupt flag x86 instruction.

**TCB** Trusted Computing Base.

**TLB** Translation Lookaside Buffer.

**TLS** Transport Layer Security.

**TPM** Trusted Platform Module.

**TRESOR** TRESOR Runs Encryption Securely Outside RAM.

**TSEG** Top Segment.

**TSS** Task State Segment.

**TZ-RKP** TrustZone-based Real-time Kernel Protection.

**URL** Uniform Resource Locator.

**US** United States.

**USB** Universal Serial Bus.

**VM** Virtual Machine.

**VRAM** Video RAM.

**WRS** WRite Select.

# Index

# Bibliography

Anderson, Ross and Markus Kuhn (1997). "Low cost attacks on tamper resistant devices". In: *International Workshop on Security Protocols*. Springer, pp. 125–136.

Anderson, Ross et al. (2006). "Cryptographic processors-a survey". In: *Proceedings of the IEEE* 94.2, pp. 357–369.

Aublin, Pierre-Louis et al. (2017). "TaLoS: Secure and transparent TLS termination inside SGX enclaves". In: *Imperial College London, Tech. Rep* 5.

Azab, Ahmed M et al. (2014). "Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, pp. 90–102.

Bajikar, Sundeep (2002). "Trusted Platform Module (TPM) based Security on Notebook PCs — White Paper". In: *Mobile Platforms Group Intel Corporation* 1, p. 20.

Ball, James, Julian Borger and Glenn Greenwald (Sept. 2013). *Revealed: how US and UK spy agencies defeat internet privacy and security*. `https://www.`

theguardian.com/world/2013/sep/05/nsa-gchq-encryption-codes-security. (Visited on 08/03/2019).

Banga, Gaurav and Peter Druschel (1999). "Measuring the capacity of a Web server under realistic loads". In: *World Wide Web* 2.1-2, pp. 69–83.

Bar-El, Hagai et al. (2006). "The sorcerer's apprentice guide to fault attacks". In: *Proceedings of the IEEE* 94.2, pp. 370–382.

Barde, Kaushik C (2014). *Hypervisor security using SMM*. US Patent 8,843,742.

Beekman, Jethro Gideon (2016). "Improving Cloud Security using Secure Enclaves". PhD thesis. UC Berkeley.

Bell, C. Gordon and Allen Newell (1971). *Computer Structures: Readings and Examples*. McGraw-Hill Inc.

Bernstein, Daniel (2013). https://twitter.com/hashbreaker/status/378258465291915264. (Visited on 08/03/2019).

Blass, Erik-Oliver and William Robertson (2012). "TRESOR-HUNT: attacking CPU-bound encryption". In: *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, pp. 71–78.

Brumley, David and Dawn Song (2004). "Privtrans: Automatically partitioning programs for privilege separation". In: *USENIX Security Symposium*, pp. 57–72.

Cambridge University Press (2018). URL: https://dictionary.cambridge.org/dictionary/english/enclave#dataset-british (visited on 08/03/2019).

Chen, Guoxing et al. (2018b). "SGXPECTRE Attacks: Leaking Enclave Secrets via Speculative Execution". In: *arXiv preprint arXiv:1802.09085*.

Chen, Guoxing et al. (2018a). "SgxPectre Attacks: Leaking Enclave Secrets via Speculative Execution". In: *CoRR* abs/1802.09085. arXiv: 1802.09085. URL: http://arxiv.org/abs/1802.09085 (visited on 08/03/2019).

Churchhouse, Robert and RF Churchhouse (2002). *Codes and ciphers: Julius Caesar, the Enigma, and the Internet*. Cambridge University Press.

Collins, Robert R. (Feb. 1996). *Undocumented OpCodes: ICEBP*. URL: http://www.rcollins.org/secrets/opcodes/ICEBP.html (visited on 06/03/2019).

— (Nov. 1997a). *ICE Mode and the Pentium Processor*. URL: http://www.rcollins.org/ddj/Nov97/Nov97.html (visited on 06/03/2019).

— (Sept. 1997b). *In-Circuit Emulation: How the Microprocessor Evolved Over Time*. URL: http://www.rcollins.org/ddj/Sep97/ (visited on 06/03/2019).

Cooke, Evan, Farnam Jahanian and Danny McPherson (2005). "The Zombie Roundup: Understanding, Detecting, and Disrupting Botnets." In: *SRUTI* 5, pp. 6–6.

Copeland, B Jack (2010). *Colossus: The secrets of Bletchley Park's code-breaking computers*. Oxford University Press.

Corbató, Fernando J, Marjorie Merwin-Daggett and Robert C Daley (1962). "An experimental time-sharing system". In: *Proceedings of the May 1-3, 1962, spring joint computer conference*. ACM, pp. 335–344.

Cranor, Charles D and Gurudatta M Parulkar (1999). "The UVM virtual memory system". In: *In Proceedings of the 1999 USENIX Annual Technical Conference*.

*CVE-2017-5689: Intel Active Management Technology Authentication Flaw Lets Remote and Local Users Gain Elevated Privileges* (2017). Available from MITRE, CVE-ID CVE-2017-5689. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5689.

*CVE-2018-1038: Microsoft Windows — Local Privilege Escalation* (2018). Available from MITRE, CVE-ID CVE-2018-1038. URL: `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1038`.

*CVE-2018-3665: Intel Core-based Processors 'Lazy FPU Restore' Lets Local Users Obtain Potentially Sensitive FPU State Information on the Target System* (2018). Available from MITRE, CVE-ID CVE-2018-3665. URL: `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3665`.

DeBusschere, Eric and Mike McCambridge (2012). "Modern game console exploitation". In: *Technical Report, Department of Computer Science, University of Arizona*.

Denning, Peter J. (Sept. 1970). "Virtual Memory". In: *ACM Comput. Surv.* 2.3, pp. 153–189. ISSN: 0360-0300. DOI: `10.1145/356571.356573`. URL: `http://doi.acm.org/10.1145/356571.356573` (visited on 08/03/2019).

Ding, Baozeng et al. (2013). "HyperVerify: a VM-assisted architecture for monitoring hypervisor non-control data". In: *Software Security and Reliability-Companion (SERE-C), 2013 IEEE 7th International Conference on*. IEEE, pp. 26–34.

Dornseif, Maximillian (2004). "0wned by an iPod". In: *Presentation, PacSec*.

Ducklin, Paul. *Sony PS3 hacked "for good" —- master keys revealed*. URL: `https://nakedsecurity.sophos.com/2012/10/25/sony-ps3-hacked-for-good-master-keys-revealed/` (visited on 26/02/2019).

Eastlake, Donald (1999). "RFC 2535: Domain name system security extensions". In: *Obsoleted by RFC4033-4035. Updated by RFC2931, RFC3007, RFC3008, RFC3090, RFC3226, RFC3445, RFC3597, RFC3655, RFC3658, RFC3755, RFC3757, RFC3845*.

Eclypsium (May 2018). *System Management Mode Speculative Execution Attacks*. URL: https://eclypsium.com/2018/05/17/system-management-mode-speculative-execution-attacks/ (visited on 08/03/2019).

edepot.com. *PlayStation 3 Secrets*. URL: http://www.edepot.com/playstation3.html (visited on 26/02/2019).

Embleton, Shawn, Sherri Sparks and Cliff C Zou (2013). "SMM rootkit: a new breed of OS independent malware". In: *Security and Communication Networks* 6.12, pp. 1590–1605.

Furtak, Andrew et al. (2014). "Bios and secure boot attacks uncovered". In: *The 10th ekoparty Security Conference*.

Ghedini, Alessandro. *Make SSL boring again*. URL: https://blog.cloudflare.com/make-ssl-boring-again/ (visited on 26/02/2019).

Gifford, David K. and John M. Lucassen (1986). "Integrating Functional and Imperative Programming". In: *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*. LFP '86. Cambridge, Massachusetts, USA: ACM, pp. 28–38. ISBN: 0-89791-200-4. DOI: 10.1145/319838.319848. URL: http://doi.acm.org/10.1145/319838.319848 (visited on 08/03/2019).

Giller, Brett (2015). "Implementing practical electrical glitching attacks". In: *Black Hat Europe*.

Gjerdrum, Anders T et al. (2017). "Performance of Trusted Computing in Cloud Infrastructures With Intel SGX". In: *Proceedings of the 7th International Conference on Cloud Computing and Services Science. Porto, Portugal: SCITEPRESS*, pp. 696–703.

Google (2013). *Coreboot SMM handler source*. URL: `https://github.com/coreboot/coreboot/blob/master/src/cpu/x86/smm/smm_module_handler.c` (visited on 06/03/2019).

Gotzfried, J and T Muller (2013). "ARMORED: CPU-bound Encryption for Android-driven ARM Devices". In: *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*. IEEE, pp. 161–168.

Graham-Cumming, John. *Incident report on memory leak caused by Cloudflare parser bug*. URL: `https://blog.cloudflare.com/incident-report-on-memory-leak-caused-by-cloudflare-parser-bug/` (visited on 26/02/2019).

Gueron, Shay (2010). "Intel® advanced encryption standard (AES) new instructions set". In: *Intel Corporation*.

Halderman, J Alex et al. (2009). "Lest we remember: cold-boot attacks on encryption keys". In: *Communications of the ACM* 52.5, pp. 91–98.

Huang, Andrew (2002). "Keeping secrets in hardware: The Microsoft Xbox TM case study". In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, pp. 213–227.

Hudson, Trammell and Larry Rudolph (2015). "Thunderstrike: EFI firmware bootkits for apple macbooks". In: *Proceedings of the 8th ACM International Systems and Storage Conference*. ACM, p. 15.

IBM (1955). *704 Electronic Data-Processing Machine Manual of Operation*. URL: `http://bitsavers.org/pdf/ibm/704/24-6661-2_704_Manual_1955.pdf` (visited on 06/03/2019).

Intel Corporation. *Intel® 64 and IA-32 ArchitecturesSoftware Developer's Manual*. URL: `https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf` (visited on 26/02/2019).

Intel Corporation. *Intel Platform Innovation Framework for EFI System Management Mode Core Interface Specification (SMM CIS) v0.9*. URL: https://www.intel.com/content/www/us/en/processors/itanium/efi-smm-cis-v09.html (visited on 26/02/2019).

Kim, Yoongu et al. (2014). "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors". In: *ACM SIGARCH Computer Architecture News*. Vol. 42 3. IEEE Press, pp. 361–372.

Kingsbury, Alex (June 2009). *The Secret History of the National Security Agency*. https://www.usnews.com/opinion/articles/2009/06/19/the-secret-history-of-the-national-security-agency. (Visited on 28/06/2018).

Kirk, Jeremy (Dec. 2013). *Report: NSA intercepts computer deliveries to plant spyware*. https://www.pcworld.com/article/2083300/report-nsa-intercepts-computer-deliveries-to-plant-spyware.html. (Visited on 08/03/2019).

Kirk, Paul L (1953). *Crime investigation; physical evidence and the police laboratory*. New York. Interscience publishers. Inc.

Kocher, Paul et al. (2018). "Spectre Attacks: Exploiting Speculative Execution". In: *CoRR* abs/1801.01203. arXiv: 1801.01203. URL: http://arxiv.org/abs/1801.01203 (visited on 08/03/2019).

Langner, R. (2011). "Stuxnet: Dissecting a Cyberwarfare Weapon". In: *IEEE Security Privacy* 9.3, pp. 49–51. ISSN: 1540-7993. DOI: 10.1109/MSP.2011.67.

Lauterbach GmbH (Nov. 2018). *Debugging via Intel DCI User's Guide*. URL: http://www2.lauterbach.com/pdf/dci_intel_user.pdf (visited on 08/03/2019).

Lawson, Nate (Jan. 2010). *How the PS3 hypervisor was hacked*. URL: https://rdist.root.org/2010/01/27/how-the-ps3-hypervisor-was-hacked/ (visited on 08/03/2019).

Lawton, Kevin P. (Sept. 1996). "Bochs: A Portable PC Emulator for Unix/X". In: *Linux J.* 1996.29es. ISSN: 1075-3583. URL: http://dl.acm.org/citation.cfm?id=326350.326357 (visited on 08/03/2019).

Leyden, John (Nov. 2017). *Hackers abusing digital certs smuggle malware past security scanners*. URL: https://www.theregister.co.uk/2017/11/01/digital_cert_abuse/ (visited on 06/03/2019).

Libreboot Project. *Libreboot*. URL: https://libreboot.org/ (visited on 26/02/2019).

Lie, David et al. (2000). "Architectural support for copy and tamper resistant software". In: *ACM SIGPLAN Notices* 35.11, pp. 168–177.

Lipp, Moritz et al. (2018). "Meltdown". In: *arXiv preprint arXiv:1801.01207*.

Liu, Fangfei et al. (2015). "Last-level cache side-channel attacks are practical". In: *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, pp. 605–622.

MachMetrics (July 2018). *Website Size: The Average Web Page Size Is More than 2MB*. URL: https://www.machmetrics.com/speed-blog/website-size-the-average-web-page-size-is-more-than-2mb-twice-the-size-of-the-average-page-just-3-years-ago/ (visited on 06/03/2019).

Manavski, Svetlin A (2007). "CUDA compatible GPU as an efficient hardware accelerator for AES cryptography". In: *2007 IEEE International Conference on Signal Processing and Communications*. IEEE, pp. 65–68.

Mansfield-Devine, Steve (2015). "The Ashley Madison affair". In: *Network Security* 2015.9, pp. 8–16.

Microsoft Corporation (Mar. 2017). *Blocking the SBP-2 driver and Thunderbolt controllers to reduce 1394 DMA and Thunderbolt DMA threats to BitLocker*. URL: `https://support.microsoft.com/en-us/help/2516445/blocking-the-sbp-2-driver-and-thunderbolt-controllers-to-reduce-1394-d` (visited on 08/03/2019).

— (Mar. 2018). *Setting Up Kernel-Mode Debugging over a 1394 Cable Manually*. URL: `https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/setting-up-a-1394-cable-connection` (visited on 08/03/2019).

Müller, Tilo, Felix C Freiling and Andreas Dewald (2011). "TRESOR Runs Encryption Securely Outside RAM." In: *USENIX Security Symposium*, pp. 17–17.

Müller, Tilo and Michael Spreitzenbarth (2013). "Frost". In: *Applied Cryptography and Network Security*. Springer, pp. 373–388.

Müller, Tilo, Benjamin Taubmann and Felix C Freiling (2012). "TreVisor". In: *Applied Cryptography and Network Security*. Springer, pp. 66–83.

Murase, Masana et al. (2009). "Effective Implementation of the Cell Broadband Engine™ Isolation Loader". In: *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, pp. 303–313.

Patel, Avadh et al. (2011). "MARSSx86: A Full System Simulator for x86 CPUs". In: *Design Automation Conference 2011 (DAC'11)*.

Poskanzer, Jef. *http_load*. URL: `{https://acme.com/software/http_load/}` (visited on 26/02/2019).

Pratt, Vaughan (1995). "Anatomy of the Pentium Bug". In: *In TAPSOFT'95: Theory and Practice of Software Development*. Springer Verlag, pp. 97–107. URL: `http://boole.stanford.edu/pub/anapent.pdf` (visited on 08/03/2019).

Provos, Niels (2000). "Encrypting Virtual Memory." In: *USENIX Security Symposium*, pp. 35–44.

Provos, Niels, Markus Friedl and Peter Honeyman (2003). "Preventing Privilege Escalation". In: *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*. SSYM'03. Washington, DC: USENIX Association, pp. 16–16. URL: `http://dl.acm.org/citation.cfm?id=1251353.1251369` (visited on 08/03/2019).

Qualys (2014). *SSL Labs SSL server test*. URL: `https://www.ssllabs.com/` (visited on 08/03/2019).

Rutkowska, Joanna and Rafał Wojtczuk (2008). "Preventing and detecting Xen hypervisor subversions". In: *Blackhat Briefings USA*.

Schwarz, Michael et al. (2017). "Malware Guard Extension: Using SGX to Conceal Cache Attacks". In: *arXiv preprint arXiv:1702.08719*.

SeaBIOS Project. *SeaBIOS*. URL: `https://www.seabios.org/SeaBIOS` (visited on 26/02/2019).

Seaborn, Mark and Thomas Dullien (2015). "Exploiting the DRAM rowhammer bug to gain kernel privileges". In: *Black Hat*, pp. 7–9.

Seo, Jaebaek et al. (2017). "SGX-Shield: Enabling address space layout randomization for SGX programs". In: *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*.

Shinagawa, Takahiro et al. (2009). "Bitvisor: a thin hypervisor for enforcing i/o device security". In: *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, pp. 121–130.

Smith, Sean W and Steve Weingart (1999). "Building a high-performance, programmable secure coprocessor". In: *Computer Networks* 31.8, pp. 831–860.

Suh, Edward et al. (2003). *Hardware mechanisms for memory authentication*. Citeseer.

Sullivan, Nick. *Keyless SSL: The Nitty Gritty Technical Details*. URL: `https://blog.cloudflare.com/keyless-ssl-the-nitty-gritty-technical-details/` (visited on 26/02/2019).

— *The Results of the CloudFlare Challenge*. URL: `https://blog.cloudflare.com/the-results-of-the-cloudflare-challenge/` (visited on 26/02/2019).

Sutherland, James, Natalie Coull and Allan MacLeod (2014). "CPU covert channel accessible from JavaScript". In: *CyberForensics 2014*.

Tereshkin, Alexander (2010). "Evil Maid Goes After PGP Whole Disk Encryption". In: *Proceedings of the 3rd International Conference on Security of Information and Networks*. SIN '10. Taganrog, Rostov-on-Don, Russian Federation: ACM, pp. 2–2. ISBN: 978-1-4503-0234-0. DOI: `10.1145/1854099.1854103`. URL: `http://doi.acm.org/10.1145/1854099.1854103` (visited on 08/03/2019).

Trent, Gene and Mark Sake (1995). *WebSTONE: The first generation in HTTP server benchmarking*.

Tzu, Sun (6th century BC). *Sun Tzu Art of War*. Vij Books India Pvt Ltd.

Ubuntu (May 2011). *[Lenovo W520] laptop freezes on ACPI-related actions*. URL: `https://bugs.launchpad.net/ubuntu/+source/linux/+bug/776999` (visited on 06/03/2019).

Ven, Adriaan van de et al. (2016). *Supervisor mode execution protection*. US Patent 9,323,533.

Wang, Jiang, Angelos Stavrou and Anup Ghosh (2010). "HyperCheck: A hardware-assisted integrity monitor". In: *Recent Advances in Intrusion Detection*. Springer, pp. 158–177.

Wang, Jiang et al. (2011). "Firmware-assisted memory acquisition and analysis tools for digital forensics". In: *Systematic Approaches to Digital Forensic Engineering (SADFE), 2011 IEEE Sixth International Workshop on*. IEEE, pp. 1–5.

Wang, Shaoqiang, DongSheng Xu and ShiLiang Yan (2010). "Analysis and application of Wireshark in TCP/IP protocol teaching". In: *E-Health Networking, Digital Ecosystems and Technologies (EDT), 2010 International Conference on*. Vol. 2. IEEE, pp. 269–272.

Wetter, Dirk (Oct. 2016). *Testing TLS/SSL encryption*. URL: `https://testssl.sh` (visited on 06/03/2019).

Wojtczuk, Rafal and Joanna Rutkowska (2009). "Attacking SMM memory via Intel CPU cache poisoning". In: *Invisible Things Lab*.

Yee, Bennet (1994). "Using secure coprocessors". PhD thesis. Citeseer.

Zeng, Hui et al. (July 2009). "MPTLsim: A Cycle-accurate, Full-system Simulator for x86-64 Multicore Architectures with Coherent Caches". In: *SIGARCH Comput. Archit. News* 37.2, pp. 2–9. ISSN: 0163-5964. DOI: `10.1145/1577129.1577132`. URL: `http://doi.acm.org/10.1145/1577129.1577132`.